lab4_logistic_regression

March 15, 2023

1 Regresia logistică

Termen de predare: 30 martie 2023, ora 23:00

Se vor folosi type annotations pentru variabile, parametrii tuturor funcțiilor, tipuri de retur. Se vor folosi docstrings pentru toate funcțiile. Neîndeplinirea acestei cerințe duce la înjumătățirea punctajului.

Se acordă doua puncte din oficiu. Fișierul va fi denumit tema2_ia_nume_prenume.ipynb. Verificați înainte de trimitere faptul ca execuția celulelor de sus în jos funcționează corespunzător. Asertiunile sunt obligatoriu a fi indeplinite. Suplimentar, puteti face si alte verificari.

Resurse utile:

[1] Cross entropy for dummies

[2] Understanding logistic regression

[3] Cross entropy log loss and intuition behind it

[4] Cross entropy (a se vedea secţiunea "Relation with log-likelihood")

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import csv
import functools
import pandas as pd
from typing import Union

plt.rc('font', **{'size' : 18})
```

[2]:
```
!pip install tableprint
import tableprint as tab
```

```
Collecting tableprint
  Downloading tableprint-0.9.1-py3-none-any.whl (6.8 kB)
Requirement already satisfied: wcwidth in
c:\users\lucian\anaconda3\envs\ia\lib\site-packages (from tableprint) (0.2.5)
Collecting future
  Downloading future-0.18.3.tar.gz (840 kB)
     ---------------------------------------- 0.0/840.9 kB ? eta -:--:--
     - ------------------------------------ 41.0/840.9 kB 960.0 kB/s eta 0:00:01
```

```
   ------ ------------------------------- 143.4/840.9 kB 1.4 MB/s eta 0:00:01
   ----------- -------------------------- 245.8/840.9 kB 1.9 MB/s eta 0:00:01
   ------------ ------------------------- 286.7/840.9 kB 1.6 MB/s eta 0:00:01
   ----------------- -------------------- 389.1/840.9 kB 1.7 MB/s eta 0:00:01
   ---------------------- --------------- 501.8/840.9 kB 1.7 MB/s eta 0:00:01
   --------------------------- ---------- 614.4/840.9 kB 1.9 MB/s eta 0:00:01
   -------------------------------- ----- 727.0/840.9 kB 2.0 MB/s eta 0:00:01
   -------------------------------------- 840.9/840.9 kB 2.0 MB/s eta 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Building wheels for collected packages: future
  Building wheel for future (setup.py): started
  Building wheel for future (setup.py): finished with status 'done'
  Created wheel for future: filename=future-0.18.3-py3-none-any.whl size=492055
sha256=db82bc866fec054089bc451866b8260b7e2bd34529bc9862167a65f48d9d82da
  Stored in directory: c:\users\lucian\appdata\local\pip\cache\wheels\bf\5d\6a\2
e53874f7ec4e2bede522385439531fafec8fafe005b5c3d1b
Successfully built future
Installing collected packages: future, tableprint
Successfully installed future-0.18.3 tableprint-0.9.1
```

1.1 Introducere

Regresia liniară 'potriveşte' o funcţie liniară (polinomială) folosind un set de date X, pe tot domeniul $\mathbb{R}$, $linreg(x) : \mathbb{R}^m \to \mathbb{R}$. Regresia logistică are acelaşi domeniu, însă codomeniul este un set mult mai restrâns, şi anume $logreg(x) : \mathbb{R}^n \to (0, 1)$. Aceasta încearcă să prezică probabilitatea ca elementul $x \in X$ să facă parte din clasa pozitivă.

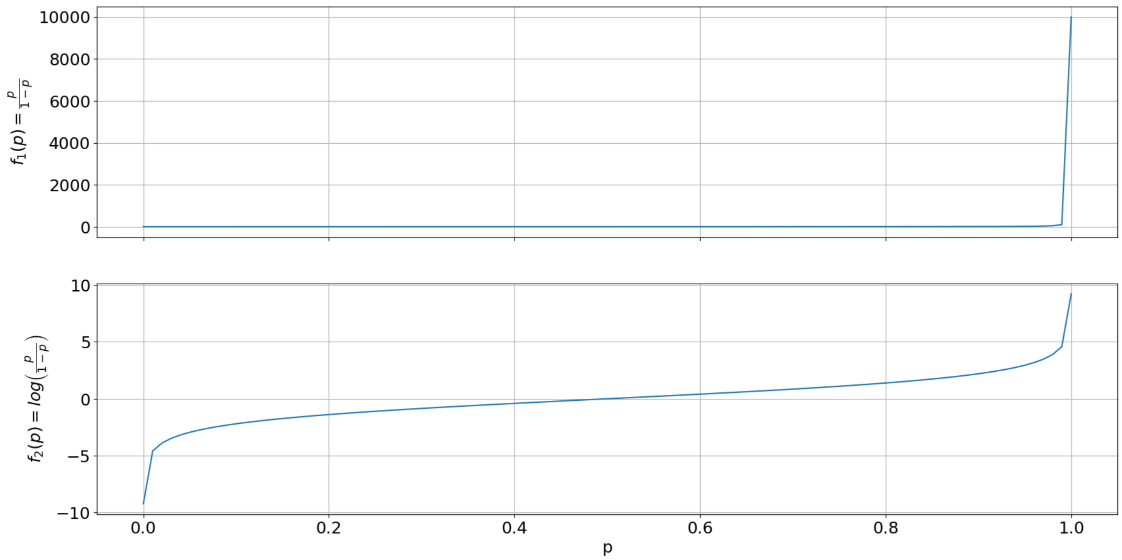
Această probabilitate o notăm cu $P(y = 1|x, \theta)$, şi o interpretăm ca fiind probabilitatea asociată răspunsului $x^T\theta$ calculat de regresia clasică, în condiţiile în care cunoaştem feature-urile X şi parametrii θ ai modelului.

Ideea este că pentru fiecare intrare x, modelul regresiei logistice asociază o probabilitate. Vom arăta cum alegem funcţia care calculează probabilitatea folosind răspunsul regresiei liniare $x^T\theta$.

Pornim de la o funcţie care are ca parametru o probabilitate şi care mapează intervalul $(0, 1)$ în toată axa reală $\mathbb{R}$. Observăm cum funcţia $f_1(p) = \frac{p}{1-p}$ mapează probabilitatea în $\mathbb{R}^+$, iar dacă aplicăm logaritmul, funcţia $f_2(p) = log\left(\frac{p}{1-p}\right)$ mapează intervalul $(0, 1)$ în toată axa reală $\mathbb{R}$:

[3]:
```python
f1 = lambda x: x / (1 - x)
f2 = lambda x: np.log(x / (1 - x))
x = np.linspace(1e-4, 1-1e-4, 100)

fig, ax = plt.subplots(2, 1, figsize=(20, 10), sharex=True)
ax[0].plot(x, f1(x)) ; ax[1].plot(x, f2(x))
ax[0].set_ylabel(r'$f_1(p) = \frac{p}{1-p}$') ; ax[0].grid()
ax[1].set_xlabel('p') ; ax[1].set_ylabel(r'$f_2(p) =␣
 ↪log\left(\frac{p}{1-p}\right)$') ; ax[1].grid()
plt.show()
```



Observaţi că dacă răspunsul dat de regresie este 0 ($f_2(p) = 0$), probabilitatea asociată este 0.5. Ne interesează ca răspunsul calculat de regresie (care poate fi orice valoare de pe axa reală) să fie egal cu această funcţie de probabilitate:

$$h_\theta(x) = x^T\theta = log\left(\frac{p}{1-p}\right)$$

Mai departe, prin câteva transformări algebrice, putem arăta că:

$$\hat{y} = P(y = 1|x, \theta) = p = \frac{1}{1 + e^{-h_\theta(x)}}$$

Avem astfel o funcţie care mapează x şi parametrii modelului θ într-o probabilitate. Funcţia este denumită funcţie logistică.

În realitate, avem setul de date X şi asociat fiecărui x_i, $i = 1 \dots m$, o valoare binară $y_i \in \{0, 1\}$. De fapt, setul de date X este caracterizat de o distribuţie de probabilitate foarte simplă, distribuţie în care probabilităţile pot lua doar valorile 0 şi 1. Dacă am pune valorile x_i pe abscisă şi valorile y_i pe ordonată, graficul distribuţiei ar fi foarte accidentat - în fond, y ia doar două valori.

Am vrea ca modelul nostru caracterizat de coeficienţii θ să se 'potrivească' cât mai bine peste această distribuţie dată iniţial - probabil peisajul funcţiei de distribuţie a modelului nu mai este atât de rugos, ci mai neted (continuu, vălurit). Bineînteles acest exemplu este unul exagerat, în realitate spaţiul de intrare X este unul m dimensional, nu 1-dimensional cum am presupus aici ca să ne imaginăm reprezentarea distribuţiilor de probabilitate.

Pentru ca să putem compara însă 'cât de bine' se potriveşte distribuţia modelului pe care îl învăţăm cu distribuţia iniţială, avem nevoie de o măsură a acestor distribuţii. Pentru aceasta vom introduce noţiunea de entropie ca măsură a informaţiei.

1.2 Regresia logistică binomială

Funcţia de cost pentru regresia logistică este dată de cross-entropia binară, scrisă sub formă vectorială astfel:

$$J(\theta) = -\frac{1}{m}\left(\mathbf{y}^t \cdot \ln \hat{\mathbf{y}} + (1 - \mathbf{y})^t \cdot \ln(\mathbb{1}_m - \hat{\mathbf{y}})\right)$$

Gradientul functiei de cost este:

$$\nabla_\theta J(\theta) = \frac{1}{m}\mathbf{X}^t(\hat{\mathbf{y}} - \mathbf{y})$$

Modificarea ponderilor din vectorul θ se face la fiecare iteratie (epoca) cu:

$$\theta = \theta - \alpha\frac{1}{m}\mathbf{X}^t(\hat{\mathbf{y}} - \mathbf{y})$$

cu $\alpha > 0$ rata de invatare.

1.2.1 Încărcarea setului de date

[4]:
```
path_train = './data/mnist_train.csv'
path_test = './data/mnist_test.csv'

train_set = pd.read_csv(path_train, header=None).values
test_set = pd.read_csv(path_test, header=None).values

assert train_set.shape == (60000, 785)
assert test_set.shape == (10000, 785)
```

[ ]:
```
# train_set si test_set sunt matrice care contin pe prima coloana clasa, daca␣
↪ca o cifra de la 0 la 9
# imaginea cifrei este pastrata incepand de pe coloana 1 pana la terminare
# Decupati in train_x doar imaginile iar in train_y doar clasa,
# si faceti acelasi lucru si pentru test_x si test_y
train_x, train_y = ...
test_x, test_y = ...

assert train_x.shape == (60000, 784)
assert train_y.shape == (60000,)
assert test_x.shape == (10000, 784)
assert test_y.shape == (10000,)
```

Să vizualizăm setul de date. Vom observa primele 16 linii din setul de antrenare:

[7]:
```
def show_samples(x_set: np.ndarray, y_set: np.ndarray) -> None:
    size = x_set.shape[0]

    fig, ax = plt.subplots(size // 5, 5, figsize=(20, 10))
```

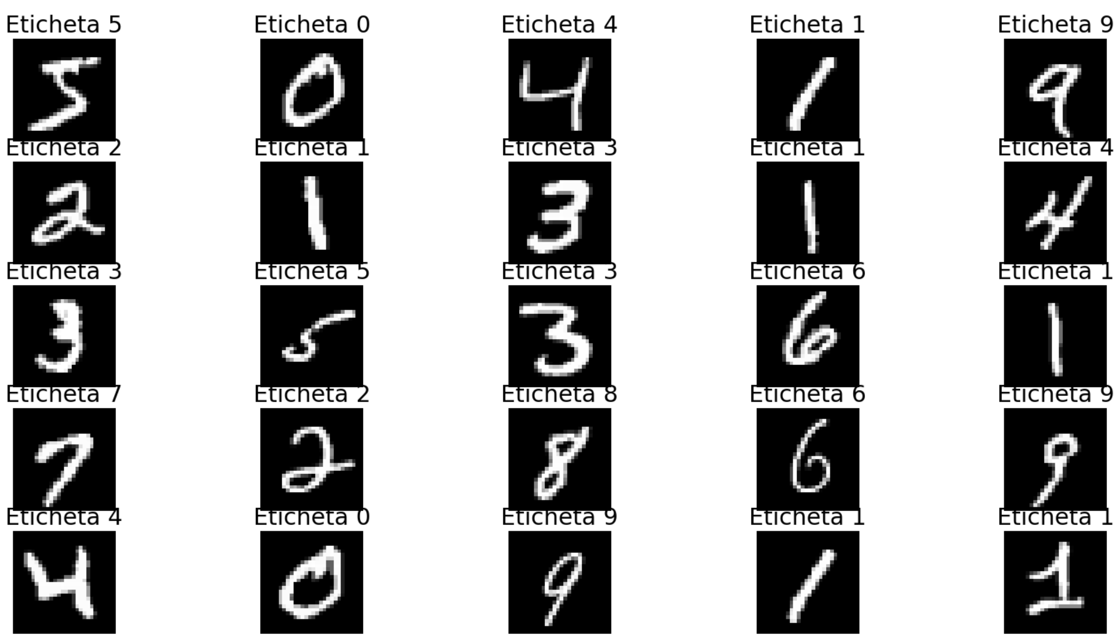
```
    for k in range(size):
        row, col = k // 5, k % 5

        # Make those columns into a array of 8-bits pixels
        # The pixel intensity values are integers from 0 to 255
        pixels = np.array(x_set[k], dtype='uint8')

        # Reshape the array into 28 x 28 array (2-dimensional array)
        n = int(np.sqrt(len(pixels)))
        assert n**2 == len(pixels)
        pixels = pixels.reshape(n, n)
        ax[row, col].imshow(pixels, cmap='gray')
        ax[row, col].set_title('Eticheta {label}'.format(label=y_set[k]))
        ax[row, col].axis('off')

    plt.show()

show_samples(train_x[:25, :], train_y[:25])
```



Pentru regresia binomială, ne interesează să clasificăm deocamdată doar imaginile corespunzătoare a două clase, de exemplu pentru cifrele '8' (clasa pozitiva) şi '4' (clasa negativa). Vom defini seturile care 'decupează' doar aceste două clase din seturile date:

[ ]:
```
# Filtrati din seturile mari doar acele sample-uri corespunzatoare cifrelor 8␣
↪si 4
# determinati vectorii logici cu True pentru indicii pt care clasele din␣
↪vectorii y sunt 8 sau 4 si False in rest
```

```
# (cifre != 8, 4)
labels_0_1_train = ...
labels_0_1_test = ...

assert labels_0_1_train.shape == (60000,)
assert labels_0_1_test.shape == (10000,)

assert labels_0_1_train.sum() == 11693
assert labels_0_1_test.sum() == 1956

# # Filtrati din seturile mari doar acele sample-uri corespunzatoare cifrelor 0␣
↪si 1.
# Puteti folosi reshape pentru vectorii y
# Folositi vectorii logici labels_0_1_train si labels_0_1_test pentru filtrare
train_x_bin, train_y_bin = ...

test_x_bin, test_y_bin = ...

assert train_x_bin.shape == (11693, 784)
assert train_y_bin.shape == (11693, 1)
assert test_x_bin.shape == (1956, 784)
assert test_y_bin.shape == (1956, 1)
```

[ ]:

```
# Momentan, vectorii 'y' sunt plini cu 8 si 4. Clasa '8' va fi clasa pozitiva␣
↪(1), clasa '4' clasa negativa (0).
# Trebuie deci sa suprascriem in cei 2 vectori 'y' valorile 8 cu 1 si 4 cu 0

# verificam ca etichetele din 'y' sunt doar 8 sau 4:
assert set(train_y_bin.flatten()) == {8, 4}
assert set(test_y_bin.flatten()) == {8, 4}

# toate etichetele de 1 vor fi trecute in 1; toate etichetele 4 vor fi trecute␣
↪in 0
train_y_bin... # setare 8 -> 1
train_y_bin... # setare 4-> 0

# toate etichetele de 1 vor fi trecute in 1; toate etichetele 4 vor fi trecute␣
↪in 0
test_y_bin... # setare 8 -> 1
test_y_bin... # setare 4-> 0

# verificam ca etichetele din 'y' sunt acum 0 sau 1:
assert set(train_y_bin.flatten()) == {0, 1}
assert set(test_y_bin.flatten()) == {0, 1}

assert train_y_bin.sum() == 5851, 'Ar trebuie sa fie 5851 exemple de clase␣
↪pozitive in setul de antrenare'
```

```
assert test_y_bin.sum() == 974, 'Ar trebuie sa fie 974 exemple de clase␣
↪pozitive in setul de testare'
```

La fel ca si la regresia liniară, prima coloană trebuie să fie formată doar din cifra 1:

[ ]:
```
def add_ones_column(x: np.ndarray) -> np.ndarray:
    """
    Returns a matrix with first column filled with 1 and the other columns␣
↪being x's columns.
    """
    # ...
    return ...

train_x_bin_ext = add_ones_column(train_x_bin)
test_x_bin_ext = add_ones_column(test_x_bin)

assert train_x_bin_ext.shape == (11693, 785)
assert test_x_bin_ext.shape == (1956, 785)
assert np.all(train_x_bin_ext[:, 0] == 1)
assert np.all(test_x_bin_ext[:, 0] == 1)
```

Trasaturile trebuie normalizate, valoarea maximă actuală fiind 255. Normalizarea urmăreşte ca toate featurile rezultate să fie în intervalul [0, 1], deci vom împărţi la valoarea maximă.

[ ]:
```
def normalize(x: np.ndarray) -> np.ndarray:
    """
    Normalization means division by 255.

    Args:
        x: feature matrix, shape m * n. It will not be changed by this code.

    Returns:
        matrix with scaled values between 0 and 1, of same shape
    """
    return ...

train_x_bin_ext = add_ones_column(normalize(train_x_bin))
test_x_bin_ext = add_ones_column(normalize(test_x_bin))

assert train_x_bin_ext.shape == (11693, 785)
assert test_x_bin_ext.shape == (1956, 785)
assert np.all(train_x_bin_ext[:, 0] == 1)
assert np.all(test_x_bin_ext[:, 0] == 1)
assert np.all(train_x_bin_ext <= 1)
assert np.all(test_x_bin_ext <= 1)
assert np.all(train_x_bin_ext >= 0)
assert np.all(test_x_bin_ext >= 0)
```

Calculăm funcţia sigmoidă, $sigmoid(z) = \frac{1}{1+e^{-z}}$ respectiv $\hat{y} = h(x, \theta) = sigmoid\,(X\theta)$:

```
[ ]: def sigmoid(z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
         return ...

     assert sigmoid(0) == 0.5
     assert np.abs(sigmoid(1) - 0.731058) < 1e-6

     def h(x: Union[float, np.ndarray], theta: Union[float, np.ndarray]) ->␣
      ↪Union[float, np.ndarray]:
         return ...

     assert np.abs(h(np.array([1., 0., 1., 1]), np.array([2., 1., 0., 0.5])) - 0.
      ↪924141) < 1e-6
```

Calculăm funcţia de cost cu regularizare de data aceasta (atenţie, coeficientul θ_0 nu se regularizează):

$$
\begin{aligned}
J(\theta) = & \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot \ln h_\theta(\mathbf{x}^{(i)}) + (1-y^{(i)}) \cdot \ln(1-h_\theta(\mathbf{x}^{(i)}))\right]}_{\text{Eroarea de calitate}} && (1)\\
+ & \underbrace{\frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2}_{\text{termenul de regularizare}} && (2)\\
= & \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot \ln \hat{y}^{(i)} + (1-y^{(i)}) \cdot \ln(1-\hat{y}^{(i)})\right]}_{\text{Eroarea de calitate}} && (3)\\
+ & \underbrace{\frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2}_{\text{termenul de regularizare}} && (4)\\
= & -\frac{1}{m}\left(\mathbf{y}^t \cdot \ln \hat{\mathbf{y}} + (\mathbb{1}-\mathbf{y})^t \cdot \ln(\mathbb{1}-\hat{\mathbf{y}})\right) + \frac{\lambda}{2}\|\theta[1:]\|_2^2 && (5)
\end{aligned}
$$

unde $\theta[1:]$ este vectorul format din toate componentele lui θ mai puțin prima, iar $\|\mathbf{v}\|_2$ este norma Euclidiană a vectorului $\mathbf{v}$.

```
[ ]: def cost(x: np.ndarray, y: np.ndarray, theta: np.ndarray, lmbda: float) ->␣
      ↪float:
         """
         Cost function includes also regularization

         Args:
             x: matricea de design a feature-urilor, dimensiune m x (n+1)
             y: vectorul ground truth, dimensiune m x 1
             theta: vectorul ponderilor modelului, dimensiune (n+1) x 1
```

```
    Returns:
        costul, ca scalar
    """
    assert x.shape[1] == theta.shape[0]
    assert x.shape[0] == y.shape[0]
    assert theta.shape[1] == y.shape[1] == 1
    assert lmbda >= 0
    # calculam y hat folosind modelul din functia h:
    y_hat = ...
    # calculam eroarea de calitate
    m = x.shape[0]
    j1 = ...
    # calculam termenul de regularizare
    j2 = ...
    return (j1 + j2).flatten()


np.random.seed(11)
n = train_x_bin_ext.shape[1]
theta = np.random.randn(n).reshape(n, 1)
assert np.abs(cost(train_x_bin_ext, train_y_bin, theta=theta, lmbda=0.2) - 80.
 ↪963789) < 1e-6
```

Calculăm gradientul, folosind expresia determinată anterior, şi ţinând seama de termenul de regularizare:

$$\nabla_\theta J(\theta) = \frac{1}{m}\mathbf{X}^t(\hat{\mathbf{y}} - \mathbf{y}) + \lambda(0, \theta_1, \dots, \theta_n)^t$$

[ ]:
```
def grad(x: np.ndarray, y: np.ndarray, theta: np.ndarray, lmbda: float) -> np.
 ↪ndarray:
    """
    Calculul gradientului pentru functia de eroare, incluzand termenul de␣
 ↪regularizare

    Args:
        x: matricea feature-urilor, dimensiune m x (n + 1)
        y: vectorul evidentei, dimensiune m
        theta: vectorul coeficientilor, dimensiune n+1
        lmbda: coeficientul de regularizare

    Returns:
        gradientul, vector de n+1 elemente
    """
    m = x.shape[0]
    y_hat = ....
```

```
    g1 = ... # gradientul termenului de calitate, fara regularizare

    theta_simple = theta.copy()
    theta_simple[0] = 0
    g2 = ... # gradientul aferent termenului de regularizare

    g = g1 + g2
    assert g.shape == theta.shape
    return g

np.random.seed(11)
n = train_x_bin_ext.shape[1]
theta = np.random.randn(n).reshape(n, 1)
res = grad(train_x_bin_ext, train_y_bin, theta=theta, lmbda=0.2)
assert res.shape == (785, 1)
```

Algoritmul de antrenare va stoca costul asociat fiecărei epoci într-o listă:

[ ]:
```
def accuracy(x_set, y_set, theta):
    pred = ((h(x_set, theta) >= 0.5) * 1 == y_set)
    return 100.0 * sum(pred) / pred.shape[0]
```

[ ]:
```
# Set learning rate
alpha = 0.2

# Set regularization coefficient
lmbda = 0.5

# In x, we have m instances of n features each
# Create theta as a vector (n x 1)
n = np.shape(train_x_bin_ext)[1]
theta = np.random.randn(n).reshape(n, 1)

# Do the training
epochs = 40
values = []
accurracies = []
for i in range(epochs):
    theta -= alpha * grad(train_x_bin_ext, train_y_bin, theta, lmbda)
    acc = accuracy(train_x_bin_ext, train_y_bin, theta)
    values.append(cost(train_x_bin_ext, train_y_bin, theta, lmbda))
    accurracies.append(acc)

print("last cost: %g" % values[-1])
```

[ ]:
```
fig, ax = plt.subplots(2, 1, figsize=(20, 10), sharex=True)
ax[0].plot(range(len(values)), values)
```

```
ax[0].set_xlabel('epoch') ; ax[0].set_ylabel('cost')
ax[0].grid()
ax[1].plot(range(len(accurracies)), accurracies)
ax[1].set_xlabel('epoch') ; ax[1].set_ylabel('accurracy [%]')
ax[1].grid()
plt.show()
```

[ ]:
```
# Calculam acuratetea pentru setul de test
# Aceasta presupune sa numaram cate predictii se potrivesc cu realitatea si sa␣
↪exprimam acest lucru procentual
pred = ((h(test_x_bin_ext, theta) >= 0.5) * 1 == test_y_bin)
print("accuracy: %2.2f%% for %d patterns" % (... , test_x_bin_ext.shape[0]))

# Calculam confusion matrix
# true positive: y = 1 and pred = 1
# true negative: y = 0 and pred = 0
# false positive: y = 0 and pred = 1
# false negative: y = 1 and pred = 0
pred = (h(test_x_bin_ext, theta) >= 0.5) * 1
tp = np.sum(np.logical_and(pred == 1, test_y_bin == 1))
tn = np.sum(np.logical_and(pred == 0, test_y_bin == 0))
fp = np.sum(np.logical_and(pred == 1, test_y_bin == 0))
fn = np.sum(np.logical_and(pred == 0, test_y_bin == 1))

headers = ['Confusion Matrix', 'pred: 0', 'pred: 1', 'pred: all']
table = [
    ['actual: 0', tn, fp, tn + fp],
    ['actual: 1', fn, tp, fn + tp],
    ['actual: all', tn + fn, fp + tp, tn + fn + fp + tp]]
tab.table(table, headers, width=16)
```

1.3 Regresia logistică multinomială

[ ]:
```
# classes
k = 10

# Adaugam la features coloana de 1-uri
train_x_all_ext = add_ones_column(normalize(train_x))
test_x_all_ext = add_ones_column(normalize(test_x))

assert train_x_all_ext.shape == (60000, 785)
assert test_x_all_ext.shape == (10000, 785)
assert np.all(train_x_all_ext[:, 0] == 1)
assert np.all(test_x_all_ext[:, 0] == 1)
assert np.all(train_x_all_ext <= 1)
assert np.all(test_x_all_ext <= 1)
assert np.all(train_x_all_ext >= 0)
```

```
assert np.all(test_x_all_ext >= 0)
```

[ ]:
```
def one_hot(val: int, classes: int) -> np.ndarray:
    """
    Realizeaza 'one-hot encoding', conversia unui intreg la un array binar,
    care are 1 doar pe pozitia specificata de val

    Args:
        val: clasa ce trebuie encodata, un intreg intre {0, 1, ... K-1}
        classes: numarul de clase K

    Returns:
        un array de zerouri de lungime K, unde doar pe pozitia val avem o␣
 ↪valoare 1
    """
    assert 0 <= val < classes
    result = ... # instructiuni
    assert result.shape == (1, classes)
    return result

assert np.all(one_hot(7, k) == np.array([[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]]))
assert np.all(one_hot(3, k) == np.array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]]))

train_y_all = np.concatenate([one_hot(int(i), k) for i in train_y])
test_y_all = np.concatenate([one_hot(int(i), k) for i in test_y])
assert train_y_all.shape == (60000, 10)
assert test_y_all.shape == (10000, 10)
assert np.all((train_y_all != 0) == (train_y_all == 1))
```

Produsul $X\theta$ între matricea X de dimensiune (m, n) şi θ de dimensiune (n, k) va avea dimensiunea (m, k):

[ ]:
```
def prod(x: np.ndarray, theta: np.ndarray) -> np.ndarray:
    """
    Product between X of shape (m x n) and theta of shape (n x k)

    Args:
        x: feature-urile, dimensiune m x n
        theta: parametrii, de dimensiune n x k

    Returns:
        produsul lor de dimensiune m x k
    """
    return ...

m, n = train_x_all_ext.shape
np.random.seed(11)
```

```python
theta = np.random.randn(n, k)
assert prod(train_x_all_ext, theta).shape == (m, k)
```

Funcţia *softmax*() va avea aceleaşi dimensiuni (m, k) şi trebuie să dea pe fiecare coloană suma 1. Se poate scrie compact calculul ei astfel:

$$softmax(X, \theta) = \frac{e^{X\theta}}{e^{X\theta} \cdot \mathbb{1}_k}$$

Termenul de la numitor, $e^{X\theta} \cdot \mathbb{1}_k$, nu mai este o matrice, ci un vector de dimensiunea (m, 1) (practic se realizează suma pe fiecare linie). Pentru realizarea împărţirii se realizează operaţia de broadcast.

[ ]:
```python
def softmax(x: np.ndarray, theta: np.ndarray) -> np.ndarray:
    """"
    Calculul functie softmax

    Args:
        x: feature-urile, dimensiune m x n
        theta: parametrii, de dimensiune n x k

    Returns:
        produsul lor de dimensiune m x k
    """
    assert x.shape[1] == theta.shape[0]
    # ... # cod...
    return ...

m, n = train_x_all_ext.shape
np.random.seed(11)
theta = np.random.randn(n, k)
smax = softmax(train_x_all_ext, theta)
assert smax.shape == (m, k)
assert np.all((smax.sum(axis=1) - 1) < 1e-12)
```

Funcţia de cost ce include regularizarea poate fi scrisă mai compact astfel (a se remarca indicele de sumare i plecand de la 1):

$$J(\theta, \lambda) = -\frac{1}{m}\mathbb{1}_m^T \{Y \odot \log[softmax(X\theta)]\mathbb{1}_k\} + \frac{\lambda}{2}\sum_{i=1}^{n-1}\sum_{j=0}^{k-1}\theta_{i,j}^2 = -\frac{1}{m}\mathbb{1}_m^T \{Y \odot \log[softmax(X\theta)]\mathbb{1}_k\} + \frac{\lambda}{2}\|\Theta[1:,:]\|_F^2$$

unde pentru o matrice A de tip $m \times n$, $\|A\|_F$ e norma Forbenius: $\|A\|_F = \sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}|a_{ij}|^2}$

[ ]:
```python
def cost(x: np.ndarray, y: np.ndarray, theta: np.ndarray, lmbda: float) ->␣
↪float:
    """
```

```
    Costul include regularizarea

    Args:
        x: feature-urile, dimensiune m x n
        y: clasele, de dimensiune m x k
        theta: parametrii, de dimensiune n x k
        lmbda: parametrul de regularizare, scalar

    Returns:
        costul, ca scalar

    """
    return ... # asigurati-va ca valoarea returnata e un scalar, nu ndarray

m, n = train_x_all_ext.shape
np.random.seed(11)
theta = np.random.randn(n, k)
assert (cost(train_x_all_ext, train_y_all, theta=theta, lmbda=0.2) - 804.
 ↪384938) < 1e-6
```

Gradientul se calculează astfel:

$$\nabla_\theta J = -\frac{1}{m} X^T \left[Y - softmax(X\theta) \right] + \lambda \left[\mathbb{0}_n, \theta_{1 \ldots k-1} \right]$$

unde matricea $\left[\mathbb{0}_n, \theta_{1 \ldots k-1} \right]$ are tot dimensiunea (n, k), ca şi θ, doar că prima linie este zero.

[ ]:
```
def deltas(x: np.ndarray, y: np.ndarray, theta: np.ndarray, lmbda: float) -> np.
 ↪ndarray:
    """
    Calculeaza gradientul

    Args:
        x: feature-urile, dimensiune m x n
        y: clasele, de dimensiune m x k
        theta: parametrii, de dimensiune n x k
        lmbda: parametrul de regularizare, scalar

    Returns:
        matricea gradientilor, de dimensiunea lui theta (n x k)
    """
    # ...
    return ...

m, n = train_x_all_ext.shape
np.random.seed(11)
theta = np.random.randn(n, k)
```

```python
grad = deltas(train_x_all_ext, train_y_all, theta=theta, lmbda=0.2)
assert grad.shape == (n, k)
assert (grad.sum() + 6.0286086) < 1e-6
```

[ ]:
```python
def calculate_accurracy(set_x, set_y, theta):
    # ...
    return 100.0 * ...
```

[ ]:
```python
# numarul de clase
k = 10

lmbda, alpha = 0.05, 0.65
m, n = train_x_all_ext.shape
np.random.seed(11)
theta = np.random.randn(n, k)

epochs = 300
values = []
accurracies = []
for i in range(epochs):
    theta -= alpha * deltas(train_x_all_ext, train_y_all, theta, lmbda)
    if i % 10 == 0:
        values.append(cost(train_x_all_ext, train_y_all, theta, lmbda))
        accurracies.append(calculate_accurracy(test_x_all_ext, test_y_all,␣
 ↪theta))
        print("epoch: ", i, "cost: ", values[-1])
        lmbda *= 0.9

print("last costs: %g" % values[-1])
```

[ ]:
```python
fig, ax = plt.subplots(2, 1, figsize=(20, 10), sharex=True)
ax[0].plot([x * 10 for x in range(len(values))], values, 'o-')
ax[0].set_xlabel('epoch') ; ax[0].set_ylabel('cost')
ax[0].grid()
ax[1].plot([x * 10 for x in range(len(accurracies))], accurracies, 'o-')
ax[1].set_xlabel('epoch') ; ax[1].set_ylabel('accurracy [%]')
ax[1].grid()
plt.show()
```

[ ]:
```python
# Calculam acuratetea pentru setul de test
# Aceasta presupune sa numaram cate predictii se potrivesc cu realitatea si sa␣
 ↪exprimam acest lucru procentual
pred = ...
actual = ...
equalities = np.sum(pred == actual)
print("Test accuracy: %2.2f%%" % ( ... , ... ))
```

[ ]:
```python
# Calculam vectorii predictiilor precum si vectorul realitatii
pred = ...
actual = ...

# Confusion matrix va avea la intersectia linie/coloana cate sample-uri din␣
↪clasa data de numarul liniei
# au fost prezise ca fiind facand parte din clasa data de numarul coloanei
conf_matrix = ...
# ...

assert len(conf_matrix) == k
assert (sum(len(row) for row in conf_matrix)) == k ** 2


headers = ['CnfMat'] + [f'pr: {x}' for x in range(k)] + ['all a']
table = []
for i in range(k):
    table.append([f'act: {i}'] + ... )
table.append( ... )

tab.table(table, headers, width=6)
```