Introducere în *NumPy* și *Matplotlib*

1. Numpy

- cea mai utilizată bibliotecă Python pentru calculul matematic
- dispune de obiecte multidimensionale (vectori, matrici) și funcții optimizate să lucreze cu acestea

Importarea bibliotecii:

```
import numpy as np
```

Vectori multidimensionali:

- initializați folosind o listă din *Python*

```
a = np.array([1, 2, 3])
print(a)            # => [1 2 3]
print(type(a))      # tipul obiectului a => <class 'numpy.ndarray'>
print(a.dtype)      # tipul elementelor din a => int32
print(a.shape)      # tuple continand lungimea lui a pe fiecare dimensiune => (3,)
print(a[0])         # acceseaza elementul avand indexul 0 => 1

b = np.array([[1, 2, 3], [4, 5, 6]])
print(b.shape)      # => (2, 3)
print(b[0][2])      # => 3
print(b[0, 2])      # => 3

c = np.asarray([[1, 2], [3, 4]])
print(type(c))      # => <class 'numpy.ndarray'>
print(c.shape)      # => (2, 2)
```

- creați folosind funcții din *NumPy*

```
zero_array = np.zeros((3, 2))          # creeaza un vector continand numai 0
print(zero_array)                      # => [[0. 0.]
                                       #     [0. 0.]
                                       #     [0. 0.]]

ones_array = np.ones((2, 2))          # creeaza un vector continand numai 1
print(ones_array)                      # => [[1. 1.]
                                       #     [1. 1.]]


constant_array = np.full((2, 2), 8)    # creeaza un vector constant
print(constant_array)                  # => [[8 8]
                                       #     [8 8]]

identity_matrix = np.eye(3)        # creeaza matricea identitate de dimensiune 3x3
print(identity_matrix)             # => [[1. 0. 0.]
                                   #     [0. 1. 0.]
                                   #     [0. 0. 1.]]
```

```
random_array = np.random.random((1,2))  # creeaza un vector cu valori aleatoare
                                        # uniform distribuite intre [0, 1)
print(random_array)                     # => ex: [[0.00672748 0.12277961]]

mu, sigma = 0, 0.1
gaussian_random = np.random.normal(mu, sigma, (3,6))# creeaza un vector cu valori
                                                    # random cu distributie
                                                    # Gaussiana de medie mu si
                                                    # deviatie standard sigma

first_5 = np.arange(5)     # creeaza un vector continand primele 5 numere naturale
print(first_5)             # => [0 1 2 3 4]
```

Indexare:

Slicing: extragerea unei submulțimi - trebuie specificați indecșii doriți pe fiecare dimensiune

```
array_to_slice = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
slice = array_to_slice[:, 0:3]      # luam toate liniile si coloanele 0, 1, 2
print(slice)                        # => [[ 1  2  3]
                                    #     [ 5  6  7]
                                    #     [ 9 10 11]]

# !! modificarea slice duce automat la modificarea array_to_slice
print(array_to_slice[0][0])         # => 1
slice[0][0] = 100
print(array_to_slice[0][0])         # => 100

# pentru a nu se intampla acest lucru submultimea poate fi copiata
slice_copy = np.copy(array_to_slice[:, 0:3])
slice_copy[0][0] = 100
print(slice_copy[0][0])        # => 100
print(array_to_slice[0][0])    # => 1
```

În cazul în care unul din indecși este un întreg, dimensiunea submulțimii returnate este mai mică decât dimensiunea inițială:

```
slice_1 = array_to_slice[2:3, :]
print(slice_1)                      # => [[ 9 10 11 12]]
slice_2 = array_to_slice[2, :]
print(slice_2)                      # => [ 9 10 11 12]

# returnarea tuturor elementelor intr-un array 1D:
slice_1d = np.ravel(slice_1)
print(slice_1d)                   # => [ 9 10 11 12]


# dimensiunea vectorilor poate fi modificata folosind functia np.reshape
reshaped_array = np.reshape(array_to_slice, (2, 6))
print(reshaped_array)           # => [[ 1  2  3  4  5  6]
                                #     [ 7  8  9 10 11 12]]
```

Folosind vectori de întregi:

```
print(array_to_slice[[0,0], [1,3]])  # afiseaza elementele de pe pozitiile
                                     # [0,1] si [0,3] => [2 4]
```

Folosind vectori de valori bool:

```
# Vrem sa afisam toate elementele mai mari decat 10 din array_to_slice
bool_idx = (array_to_slice > 10)  # rezulta o matrice de aceeasi dimensiune cu
                                  # array_to_slice in care fiecare element consta
                                  # intr-o valoare bool astfel:
                                  # True, daca elementul corespunzatoe din
                                  #       array_to_slice > 10
                                  # False, daca elementul corespunzatoe din
                                  #       array_to_slice <= 10

print(bool_idx)     # => [[ True False False False]
                    #     [False False False False]
                    #     [False False  True  True]]

print(array_to_slice[bool_idx])  # => [100  11  12]

# Operatia se poate face si direct:
print(array_to_slice[array_to_slice > 10])     # => [100  11  12]
```

Funcții matematice:

Operațiile matematice de bază sunt disponibile atât ca funcții NumPy cât și ca operatori. Acestea sunt aplicate element cu element:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Suma element cu element => [[ 6.0  8.0]
#                             [ 10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Diferenta element cu element => [[ -4.0 -4.0]
#                                  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Produs element cu element => [[ 5.0  12.0]
#                               [ 21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Impartire element cu element => [[ 0.2        0.33333333]
#                                  [ 0.42857143 0.5]]
print(x / y)
print(np.divide(x, y))

# Radical element cu element => [[ 1.          1.41421356]
#                                [ 1.73205081  2.]]
```

```
print(np.sqrt(x))

# Ridicare la putere
my_array = np.arange(5)
powered = np.power(my_array, 3)
print(powered)                       # => [ 0  1  8 27 64]
```

Produsul scalar:

```
x = np.array([[1, 2],[3, 4]])
y = np.array([[5, 6],[7, 8]])

v = np.array([9, 10])
w = np.array([11, 12])

# vector x vector => 219
print(v.dot(w))
print(np.dot(v, w))

# matrice x vector => [29 67]
print(np.matmul(x, v))

# matrice x matrice => [[19 22]
#                        [43 50]]
print(np.matmul(x, y))
```

Operații pe matrici:

```
# transpusa unei matrici
my_array = np.array([[1, 2, 3], [4, 5, 6]])    # [[1, 2, 3],
                                                #  [4, 5, 6]]
print(my_array.T)             # => [[1, 4],
                              #     [2, 5],
                              #     [3, 6]]
# inversa unei matrici
my_array = np.array([[1., 2.], [3., 4.]])
print(np.linalg.inv(my_array))          # => [[-2. ,  1. ],
                                              [ 1.5, -0.5]]
```

NumPy dispune de funcții care realizează operații pe o anumită dimensiune.

```
x = np.array([[1, 2],[3, 4]])

# suma pe o anumita dimensiune
print(np.sum(x))                 # Suma tuturor elementelor => 10
print(np.sum(x, axis=0))         # Suma pe coloane => [4 6]
print(np.sum(x, axis=1))         # Suma pe linii => [3 7]
# putem specifica si mai multe axe pe care sa se faca operatia:
print(np.sum(x, axis=(0, 1)))     # Suma tuturor elementelor => 10
```

```
# media pe o anumita dimensiune
y = np.array([[[1, 2, 3, 4], [5, 6, 7, 8]], [[1, 2, 3, 4], [5, 6, 7, 8]], [[1, 2, 3,
4], [5, 6, 7, 8]]])
print(y.shape)      # => (3, 2, 4)
print(y)            # => [[[1 2 3 4]
                    #      [5 6 7 8]]
                    #     [[1 2 3 4]
                    #      [5 6 7 8]]
                    #     [[1 2 3 4]
                    #      [5 6 7 8]]]

print(np.mean(y, axis=0))       # => [[1. 2. 3. 4.]
                                #     [5. 6. 7. 8.]]

print(np.mean(y, axis=1))       # => [[3. 4. 5. 6.]
                                #     [3. 4. 5. 6.]
                                #     [3. 4. 5. 6.]]

# indexul elementului maxim pe fiecare linie
z = np.array([[10, 12, 5], [17, 11 ,19]])
print(np.argmax(z, axis=1))    # => [1 2]
```

Broadcasting:

- mecanism care oferă posibilitatea de a realiza operații aritmetice între vectori de dimensiuni diferite
- vectorul mai mic este multipilcat astfel încât să se potrivească cu cel mai mare, operația fiind apoi realizată pe cel din urmă

```
# Vrem sa adunam un vector (v) la fiecare linie a unei matrici (m)
m = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = m + v
print(y)  # => [[ 2  2  4]
          #     [ 5  5  7]
          #     [ 8  8 10]
          #     [11 11 13]]
```

Reguli de broadcasting:

1. Daca vectorii nu au același număr de dimensiuni, vectorul mai mic este extins cu câte o dimensiune, până când acest lucru este realizat.

   ex: Dacă avem 2 vectori ***a*** și ***b*** cu

   a.shape = (3, 4)

   b.shape = (6,)

   b este extins la dimensiunea (6, 1)

2. Cei 2 vectori se numesc compatibili pe o dimensiune dacă au aceeași lungime pe acea dimensiune sau dacă unul dintre ei are lungimea 1.

   ex: Considerăm vectorii:

   a astfel încât: a.shape = (3, 4)
   b astfel încât: b.shape = (6, 1)
   c astfel încât: c.shape = (3, 5)

   a și ***c*** sunt compatibili pe prima dimensiune
   a și ***b*** sunt compatibili pe cea de-a doua dimensiune

3. Pe cei 2 vectori se poate aplica broadcasting daca ei sunt compatibili pe toate dimensiunile.

4. La broadcasting, fiecare vector se comportă ca și cum ar avea, pe fiecare dimensiune, lungimea maximă dintre cele două dimensiuni inițiale (maximul dimensiunilor element cu element)

   ex: La broadcasting vectorii ***a*** și ***b*** cu
   a.shape = (3, 4)
   b.shape = (3, 1)
   se comportă ca și cum ar avea dimensiunea (3, 4)

5. În fiecare dimensiune pe care unul din vectori avea dimensiunea 1, iar celalalt mai mare, primul vector se comportă ca și cum ar fi copiat de-a lungul acelei dimensiuni.

   ex: Considerăm vectorii:

   a = [[1, 2, 3],
   [4, 5, 6]] , a.shape = (2, 3)

   b = [[1.],
   [1.]] , b.shape = (2, 1)

   Când vrem să facem o operație de broadcasting, vectorul b va fi copiat de-a lungul celei de-a doua dimensiuni, astfel încât el devine:

   b = [[1., 1., 1.],
   [1., 1., 1.]] , b.shape = (2, 3)

   operația fiind acum realizată pe vectori de aceeași dimensiune.

2. Matplotlib

- bibliotecă utilizată pentru plotarea datelor

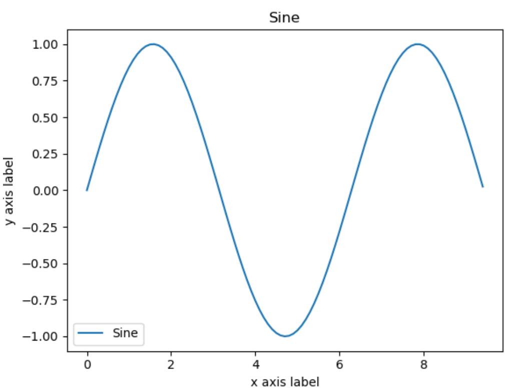
Importarea bibliotecii:

```
import matplotlib.pyplot as plt
```

Plotare:

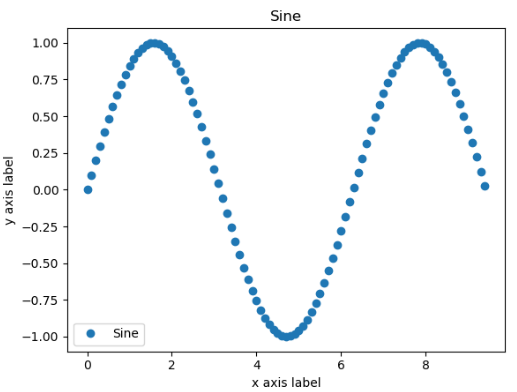
- cea mai importantă funcție este *plot*, care permite afișarea datelor 2D

```
# Calculeaza coordonatele (x, y) ale punctelor de pe o curba sin
# x - valori de la 0 la 3 * np.pi, luate din 0.1 in 0.1
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Ploteaza punctele
plt.plot(x, y)

# Adauga etichete pentru fiecare axa
plt.xlabel('x axis label')
plt.ylabel('y axis label')

# Adauga titlu
plt.title('Sine')

# Adauga legenda
plt.legend(['Sine'])

# Afiseaza figura
plt.show()
```



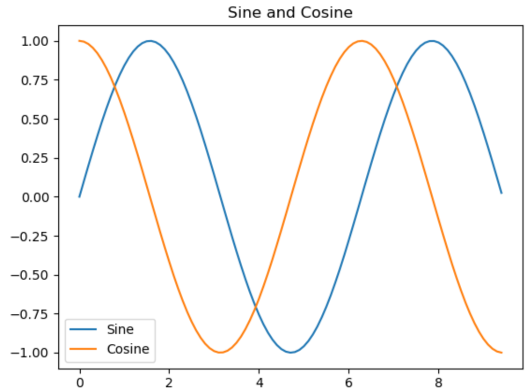
OBS. Pentru a plota punctele independent, fără a face interpolare ca în exemplul anterior, se poate specifica un al treilea parametru în funcția plot, astfel:

```
plt.plot(x, y, 'o')
```



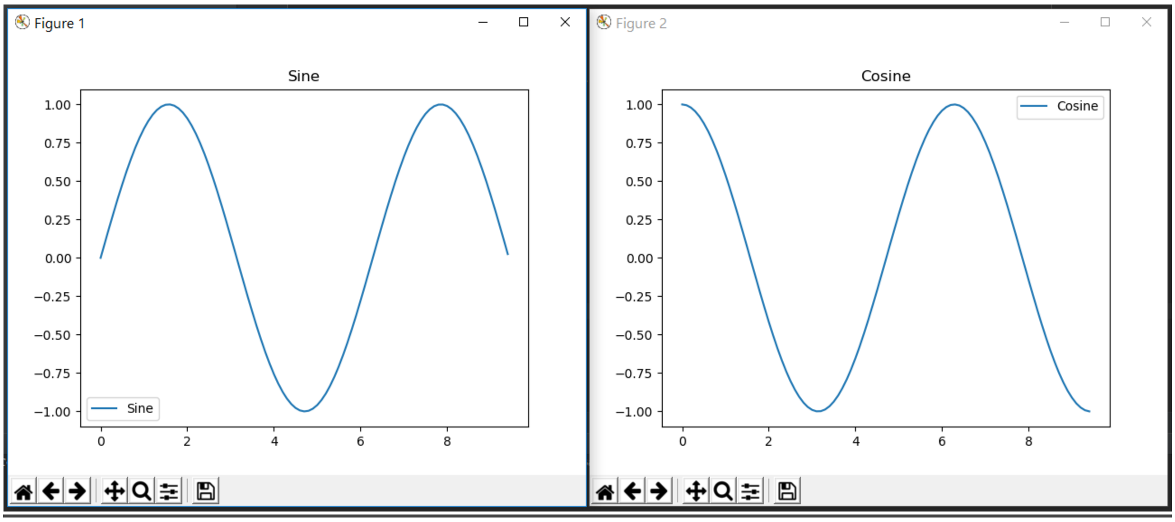
Plotarea mai multor grafice în cadrul aceleiași figuri:

```
# Calculeaza coordonatele (x, y) ale punctelor de pe o curba sin, respectiv cos
# x - valori de la 0 la 3 * np.pi, luate din 0.1 in 0.1
x = np.arange(0, 3 * np.pi, 0.1)
y_1 = np.sin(x)
y_2 = np.cos(x)

# Ploteaza punctele in aceeasi figura
plt.plot(x, y_1)
plt.plot(x, y_2)

# Adauga titlu
plt.title('Sine and Cosine')

# Adauga legenda
plt.legend(['Sine', 'Cosine'])

# Afiseaza figura
plt.show()
```



Plotarea simultană a mai multor grafice în figuri diferite:

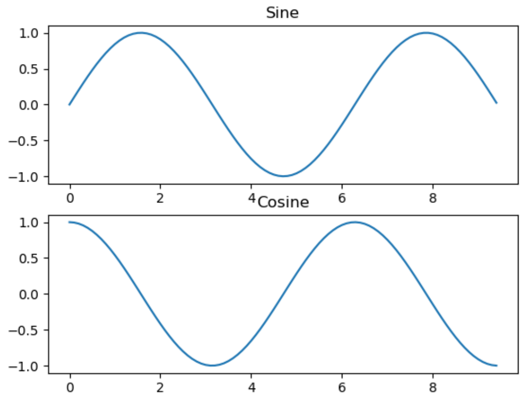
```
# Calculeaza coordonatele (x, y) ale punctelor de pe o curba sin, respectiv cos
# x - valori de la 0 la 3 * np.pi, luate din 0.1 in 0.1
x = np.arange(0, 3 * np.pi, 0.1)
y_1 = np.sin(x)
y_2 = np.cos(x)

# definim primul plot in figura 1
first_plot = plt.figure(1)
plt.plot(x, y_1)
plt.title('Sine')
plt.legend(['Sine'])

# definim cel de-al doilea plot in figura 2
second_plot = plt.figure(2)
plt.plot(x, y_2)
plt.title('Cosine')
plt.legend(['Cosine'])

# afisam figurile
plt.show()
```



Sublotare:

- putem plota mai multe subfiguri în cadrul aceleiași figuri

```
# Calculeaza coordonatele (x, y) ale punctelor de pe o curba sin, respectiv cos
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Creeaza un grid avand inaltimea 2 si latimea 1
# si seteaza primul subplot ca activ
plt.subplot(2, 1, 1)

# Ploteaza primele valori
plt.plot(x, y_sin)
plt.title('Sine')

# Seteaza cel de-al doilea subplot ca activ
# si ploteaza al doilea set de date
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Afiseaza figura
plt.show()
```



Exerciții

1. Se dau următoarele 9 imagini de dimensiuni 400x600. Valorile acestora au fost salvate în fișierele “images/car_{idx}.npy”.

   a. Citiți imaginile din aceste fișiere și salvați-le într-un np.array (va avea dimensiunea 9x400x600).

   Obs: Citirea din fișier se face cu ajutorul funcției:

```
image = np.load(file_path)
```

   Aceasta întoarce un np.array de dimensiune 400x600.

   b. Calculați suma valorilor pixelilor tuturor imaginilor.

   c. Calculați suma valorilor pixelilor pentru fiecare imagine în parte.

   d. Afișați indexul imaginii cu suma maximă.

e. Calculați imaginea medie și afișați-o.
Obs: Afișarea imaginii medii se poate face folosind biblioteca *scikit-image* în următorul mod:

```
from skimage import io
io.imshow(mean_image.astype(np.uint8)) # petru a putea fi afisata
                                       # imaginea trebuie sa aiba
                                       # tipul unsigned int
io.show()
```

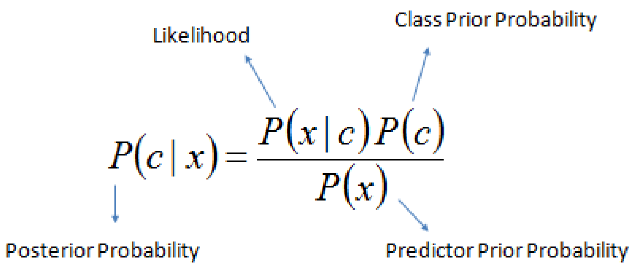
Dacă biblioteca nu este instalată, acest lucru se poate face prin rularea comenzii sistem `pip install scikit-image`.

f. Cu ajutorul funcției `np.std(images_array)`, calculați deviația standard a imaginilor.

g. Normalizați imaginile. (se scade imaginea medie și se împarte rezultatul la deviația standard)

h. Decupați fiecare imagine, afișând numai liniile cuprinse între 200 și 300, respectiv coloanele cuprinse între 280 și 400.

Naive Bayes

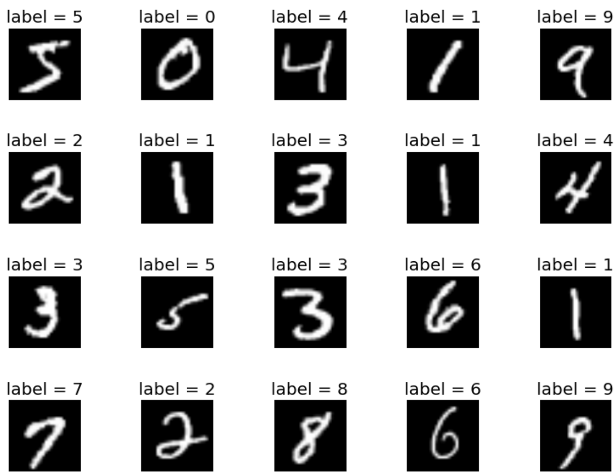


Regula Bayes

În acest laborator vom clasifica cifrele scrise de mână din subsetul **MNIST** folosind Naive Bayes.

MNIST[1] este o bază de date cu cifre scrise de mână (0-9), conținând 60.000 de imagini pentru antrenare și 10.000 pentru testare. Imaginile sunt alb-negru având dimensiunea de 28x28 pixeli. În cadrul laboratorului vom lucra pe un subset, împărțit astfel:

- ➔ În 'train_images.txt' sunt 1.000 de imagini din mulțimea de antrenare, fiecare fiind stocată pe câte o linie a matricei de dimensiune 1000 x 784 (28 x 28 = 784).
- ➔ În 'test_images.txt' sunt 500 de imagini din setul de testare.
- ➔ Fișierele 'train_labels.txt' și 'test_labels.txt' conțin etichetele imaginilor.



Exemple de imagini din setul de date MNIST.

Descărcați arhiva care conține datele de antrenare și testare [de aici.](#)

[1] http://yann.lecun.com/exdb/mnist/

Pentru vizualizarea unei imagini din mulțimea de antrenare trebuie să redimensionăm vectorul de 1 x 784 la 28 x 28.

```
import numpy as np
import matplotlib.pyplot as plt

train_images = np.loadtxt('train_images.txt') # incarcam imaginile
train_labels = np.loadtxt('train_labels.txt', 'int')  # incarcam etichetele avand
                                                      # tipul de date int
image = train_images[0, :] # prima imagine
image = np.reshape(image, (28, 28))
plt.imshow(image.astype(np.uint8), cmap='gray')
plt.show()
```

Deoarece datele noastre (valorile pixelilor) sunt valori continue, va trebui sa le transformăm în valori discrete cu ajutorul unei histograme. Vom stabili numărul de intervale la care vom împărți lungimea intervalului valorilor continue, apoi vom asigna fiecărei valori continue indicele intervalul corespunzător.

```
bins = np.linspace(start=0, stop=255, num=num_bins)  # returneaza intervalele
x_to_bins = np.digitize(x, bins) # returneaza pentru fiecare element intervalul
                                 # corespunzator
                                 # Atentie! In cazul nostru indexarea elementelor va
                                 # incepe de la 1, intrucat nu avem valori < 0
```

1. Antrenarea clasificatorului (fit)

O imagine $X = \{x_1, x_2, \ldots, x_{784}\}$ din mulțimea de antrenare are dimensiunea de 1x784. Conform presupunerii clasificatorului Naive Bayes, vom considera fiecare pixel ca fiind un atribut ***independent*** în calcularea probabilității apartenenței lui X la clasa c.

$$P(c|X) = p(c) \prod_{i=1}^{784} P(x_i|c) \quad |\ aplicăm\ logaritm$$

$$log(P(c|X)) \simeq log(P(c)) + \sum_{i=1}^{784} log(P(x_i|c))$$

Pentru aplicarea regulii Naive Bayes avem nevoie de:

1. $P(c) = \frac{numărul\ exemplelor\ din\ clasa\ c}{numărul\ total\ de\ exemple}$ $,\ probabilitatea\ ca\ un\ exemplu\ să\ se\ afle\ în\ clasa\ c$
2. $P(x|c) = \frac{numărul\ exemplelor\ din\ clasa\ c\ care\ sunt\ egale\ cu\ x}{numărul\ exemplelor\ din\ clasa\ c}$ $,\ probabilitatea\ de\ a\ avea\ atributul\ x\ în\ clasa\ c$

2. Prezicerea etichetelor pe baza clasificatorul (predict)

$$P(c|X) \ = \ p(c) \prod_{i=1}^{n} P(x_i|c) \ ,$$

$unde\ X \ = \{x_1, x_2, ..., x_n\}\ cu\ x_1, ..., x_n\ atribute\ independente.$

Probabilitatea ca exemplul $X = \{x_1, x_2, \ldots, x_{784}\}$ să fie în clasa c, se obține prin înmulțirea (sau adunarea logaritmilor) probabilităților individuale ale atributelor acestuia condiționate de clasa c. Vom calcula P(c|X) pentru fiecare clasă c (c∈ [1, num_classes]), iar eticheta finală este dată de clasa cu probabilitatea cea mai mare.

Biblioteca Scikit-learn

În continuare vom folosi biblioteca ***Scikit-learn***. Aceasta este dezvoltată în Python, fiind integrată cu NumPy și pune la dispoziție o serie de algoritmi optimizați pentru probleme de clasificare, regresie și clusterizare.

Pas 1: Instalarea librăriei

pip install scikit-learn

Pas 2: Importarea modelului

from sklearn.naive_bayes import MultinomialNB

Pas 3: Definirea modelului

naive_bayes_model = MultinomialNB()

Pas 4: Antrenarea modelului

naive_bayes_model.fit(*training_data*, *training_labels*)

Pas 5.1: Prezicerea etichetelor

naive_bayes_model.predict(*testing_data*)

Pas 5.2: Calcularea acurateții

naive_bayes_model.score(*testing_data, testing_labels*)

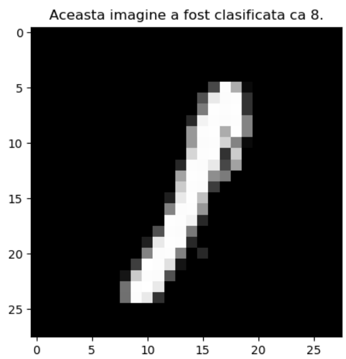
Exerciții

1. Se dă mulțimea de antrenare, reprezentând înălțimea în cm a unei persoane și eticheta corespunzătoare:
[(160, F), (165, F), (155, F), (172, F), (175, B), (180, B), (177, B), (190, B)].
Împărțind valorile continue (înălțimea) în 4 intervale (150-160, 161-170, 171-180, 181-190), calculați probabilitatea ca o persoană având 178 cm, să fie fată sau să fie băiat, folosind regula lui Bayes.

2. Știind că valoarea minimă a unui pixel este *0*, iar valoarea maximă este *255,* calculați capetele a *num_bins* intervale (utilizați funcția linspace). Definiți metoda *values_to_bins* care primește o matrice de dimensiune (n_samples, n_features) și capetele intervalelor calculate anterior, iar pentru fiecare exemplu și fiecare atribut calculează indexul intervalului corespunzător (utilizați funcția np.digitize).
Folosiți funcția definită pentru a discretiza mulțimea de antrenare și cea de testare.

3. Calculați acuratețea pe *mulțimea de testare* a clasificatorul Multinomial Naive Bayes, împărțind intervalul pixelilor în 4 sub-intervale.
OBS. Acuratețea pe care trebuie să o obțineți pentru num_bins = 5 este de 83.6%.

4. Testați clasificatorul Multinomial Naive Bayes pe subsetul MNIST folosind $num\_bins \in \{3,\ 5,\ 7,\ 9,\ 11\}$.
5. Folosind numărul de sub-intervale care obține cea mai bună acuratețe la exercițiul anterior, afișați cel puțin 10 exemple misclasate.



6. Definiți metoda *confusion_matrix(y_true, y_pred)* care calculează matricea de confuzie. Calculați matricea de confuzie folosind predicțiile clasificatorului anterior.
Obs:
- Matrice de confuzie *C* = c_{ij}, numărul exemplelor din clasa *i* care au fost clasificata ca fiind în clasa *j*.

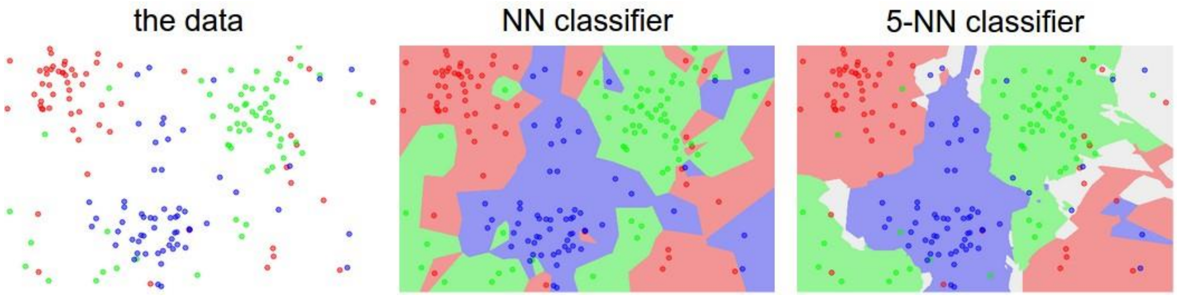
Clasa actuală↓ Clasa prezisă →	1	2	3
1	Nr. exemplelor din clasa 1 care au fost clasificate ca fiind in	Nr. exemplelor din clasa 1 care au fost clasificate ca	Nr. exemplelor din clasa 1 care au fost clasificate ca fiind in

	clasa 1	fiind in clasa 2	clasa 3
2	Nr. exemplelor din clasa 2 care au fost clasificate ca fiind in clasa 1	Nr. exemplelor din clasa 2 care au fost clasificate ca fiind in clasa 2	Nr. exemplelor din clasa 2 care au fost clasificate ca fiind in clasa 3
3	Nr. exemplelor din clasa 3 care au fost clasificate ca fiind in clasa 1	Nr. exemplelor din clasa 3 care au fost clasificate ca fiind in clasa 2	Nr. exemplelor din clasa 3 care au fost clasificate ca fiind in clasa 3

- Matricea de confuzie pentru clasificatorul anterior este:

```
[[51.  0.  0.  0.  0.  0.  1.  0.  1.  0.]
 [ 0. 48.  0.  0.  0.  0.  0.  0.  4.  0.]
 [ 2.  1. 50.  1.  1.  0.  1.  1.  1.  0.]
 [ 0.  0.  1. 49.  0.  0.  0.  0.  0.  3.]
 [ 0.  0.  0.  0. 33.  0.  0.  0.  2. 11.]
 [ 1.  0.  0.  9.  0. 34.  1.  0.  6.  1.]
 [ 1.  1.  0.  0.  1.  0. 43.  0.  2.  0.]
 [ 0.  1.  0.  0.  2.  0.  0. 41.  0.  6.]
 [ 0.  1.  3.  3.  1.  1.  1.  1. 34.  1.]
 [ 0.  0.  1.  1.  5.  0.  0.  0.  0. 35.]]
```

Metoda celor mai apropiați vecini



Exemplu care arată diferențele dintre metoda celui mai apropiat vecin și metoda celor mai apropiați cinci vecini. Zona colorată reprezintă regiunea de decizie a clasificatorului folosind distanța L2. Se observă că în cazul metodei celui mai apropiat vecin se formează mici 'insule' ce pot duce la predicții incorecte. Zonele gri din imaginea 5-NN reprezintă zone de predicție ambigue din cauza egalității voturilor celor mai apropiați vecini.

În acest laborator vom clasifica cifrele scrise de mână din subsetul **MNIST** folosind metoda celor mai apropiați vecini.

Descărcați arhiva cu datele de antrenare și testare [de aici.]()

Care este acuratețea metodei *celui* mai apropiat vecin pe mulțimea de *antrenare* când se folosește distanța L2? Dar pentru distanța L1?

Care este acuratețea metodei celor mai apropiați vecini pe mulțimea de *antrenare* când se folosește numărul de vecini $K \geq 2$ și distanța L2? Dar pentru distanța L1?

Exerciții

1. Creați clasa KnnClassifier, având constructorul următor:

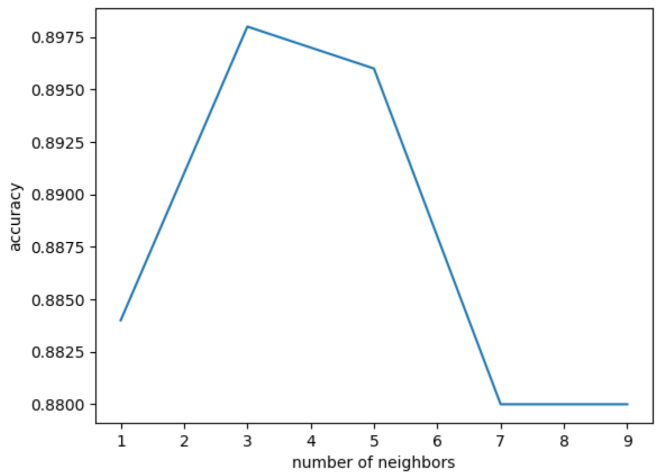
```
def __init__(self, train_images, train_labels):
    self.train_images = train_images
    self.train_labels = train_labels
```

2. Definiți metoda *classify_image(self, test_image, num_neighbors = 3, metric = 'l2')* care clasifică imaginea *test_image* cu metoda celor mai apropiați vecini, numărul vecinilor este stabilit de parametru *num_neighbors*, iar distanța poate fi L1 sau L2, în funcție de parametrul *metric*.
Obs:
    - $L1(X,Y) = \sum_{i=1}^{n} |X_i - Y_i|$

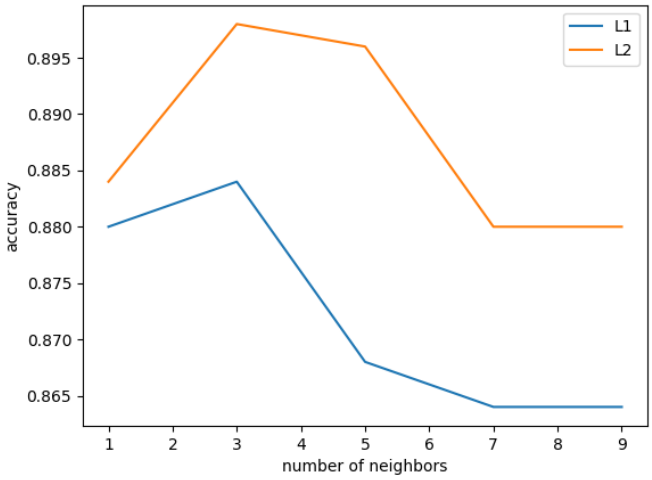
- $L2(X, Y) = \sqrt{\sum_{i=1}^{n} (X_i - Y_i)^2}$
- În variabilele *train_images* și *test_image* valorile unui exemplu sunt stocate pe linie. (train_images.shape = (num_samples, num_features), test_image.shape = (1, num_features))

3. Calculați acuratețea metodei celor mai apropiați vecini pe mulțimea de testare având ca distanța 'l2' și numărul de vecini 3. Salvați predicțiile în fișierul *predictii_3nn_l2_mnist.txt*.
   Obs:
   - Acuratețea pe mulțimea de testare este de 89.8%.

4. Calculați acuratețea metodei celor mai apropiați vecini pe mulțimea de testare având ca distanța L2 și numărul de vecini ∊ [1, 3, 5, 7, 9].
   a. Plotați un grafic cu acuratețea obținuta pentru fiecare vecin și salvați scorurile în fișierul *acuratete_l2.txt*.



   b. Repetați punctul anterior pentru distanța L1. Plotați graficul de la punctul anterior în aceeași figură cu graficul curent (utilizați fișierul *acuratete_l2.txt*).



Funcții numpy:

```
np.sort(x)  # sorteaza array-ul
np.argsort(x)  # returneaza indecsi care sorteaza array-ul
np.bincount(x) # calculeaza numarul de aparatii al fiecarei valori din array
print(np.bincount(numpy.array([0, 1, 1, 3, 2, 1, 7]))) # array([1, 3, 1, 1, 0, 0, 0, 1])
np.where(x == 3) # returneaza indecsi care satisfac conditia
np.intersect1d(x, y) # returneaza intersectia celor 2 array
np.savetxt('fisier.txt', y) # salveaza array-ul y in fisierul fisier.txt
```

Modelul bag-of-words. Normalizarea datelor. Mașini cu vectori suport (SVM).

1. Modelul bag-of-words

➔ este o metodă de reprezentare a datelor de tip text, bazată pe frecvența de apariție a cuvintelor în cadrul documentelor

➔ algoritmul este alcătuit din 2 pași:

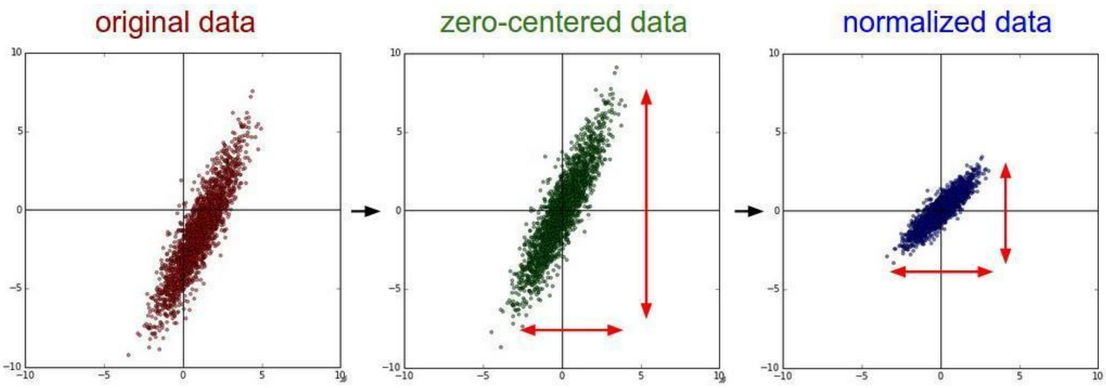
1. definirea unui vocabular prin atribuirea unui id unic fiecărui cuvânt regăsit în setul de date (setul de antrenare)

2. reprezentarea fiecărui document ca un vector de dimensiune egală cu lungimea vocabularului, definit astfel:

$$features[word\_idx] = numărul\ de\ apariții\ al\ cuvântului\ cu\ id - ul\ word\_idx$$

2. Normalizarea datelor

2.1. Standardizarea



Metode obișnuite de preprocesare a datelor. În partea **stângă** sunt reprezentate datele 2D originale. În **mijloc** acestea sunt centrate în origine, prin scăderea mediei pe fiecare dimensiune. În partea **dreaptă** fiecare dimensiune este scalată folosind deviația standard corespunzătoare. Spre deosebire de imaginea din centru, unde datele au lungimi diferite pe cele două axe, aici ele sunt egale.

- transformă vectorii de caracteristici astfel încât fiecare să aibă medie 0 și deviație standard 1

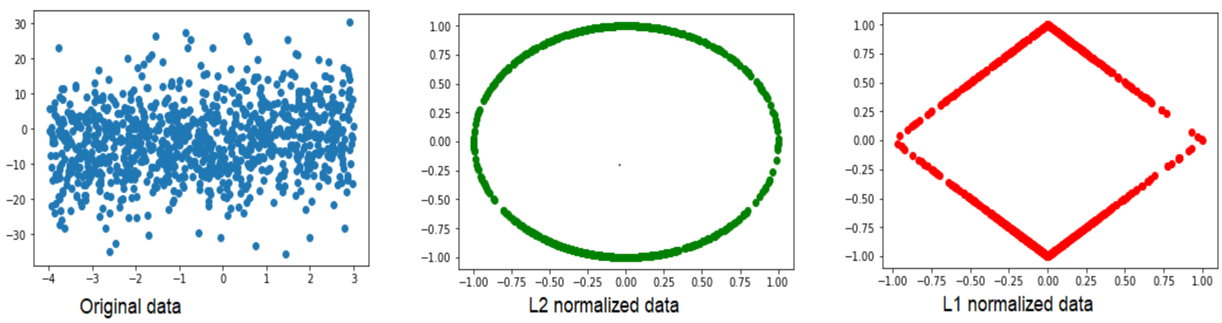
$x\_scaled = \frac{x - mean(x)}{\sigma}$, unde x_mean- media valorilor lui x

σ - deviația standard

```
from sklearn import preprocessing
import numpy as np
```

```
x_train = np.array([[1, -1, 2], [2, 0, 0], [0, 1, -1]], dtype=np.float64)
x_test = np.array([[-1, 1, 0]], dtype=np.float64)

# facem statisticile pe datele de antrenare
scaler = preprocessing.StandardScaler()
scaler.fit(x_train)

# afisam media
print(scaler.mean_)                    # => [1.  0.  0.33333333]
# afisam deviatia standard
print(scaler.scale_)                   # => [0.81649658 0.81649658 1.24721913]

# scalam datele de antrenare
scaled_x_train = scaler.transform(x_train)
print(scaled_x_train)          # => [[0.          -1.22474487  1.33630621]
                               #     [1.22474487   0.          -0.26726124]
                               #     [-1.22474487  1.22474487  -1.06904497]]

# scalam datele de test
scaled_x_test = scaler.transform(x_test)
print(scaled_x_test)         # => [[-2.44948974  1.22474487 -0.26726124]]
```

2.2. Normalizarea L1. Normalizarea L2



În partea **stângă** sunt reprezentate datele 2D originale. În **mijloc,** sunt reprezentate datele normalizate folosind norma L_2. În partea **dreaptă** , sunt reprezentate datele normalizate folosind norma L_1.

- scalarea individuală a vectorilor de caracteristici corespunzători fiecărui exemplu astfel încât norma lor să devină 1.

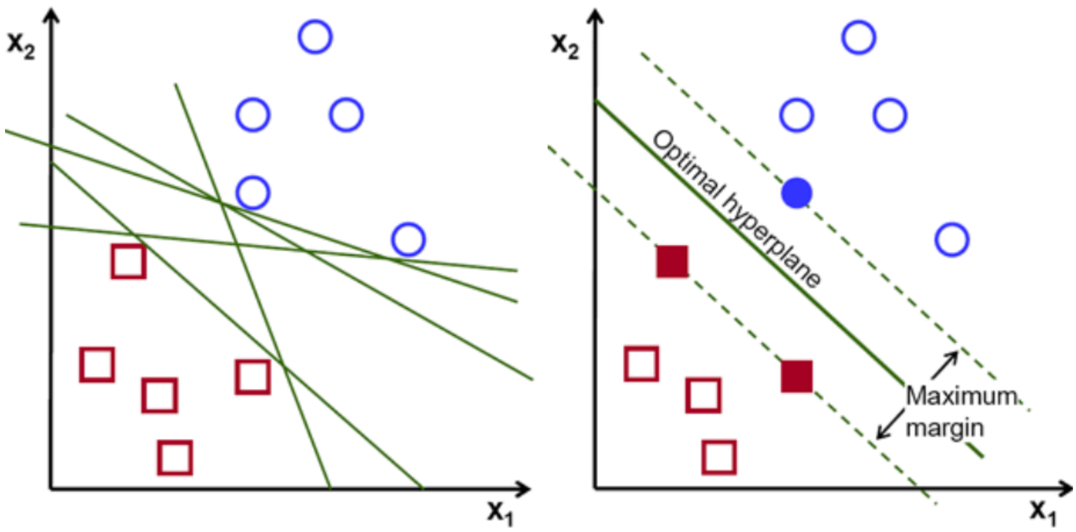
Folosind norma L_1:

$$x\_scaled = \frac{X}{||X||_1}, \quad ||X||_1 = \sum_{i=1}^{n} |x_i|$$

Folosind norma L_2:

$$x\_scaled = \frac{X}{||X||_2}, \; ||X||_2 = \sqrt{\sum_{i=1}^{n} x_i^{\;2}}$$

3. Mașini cu vectori suport



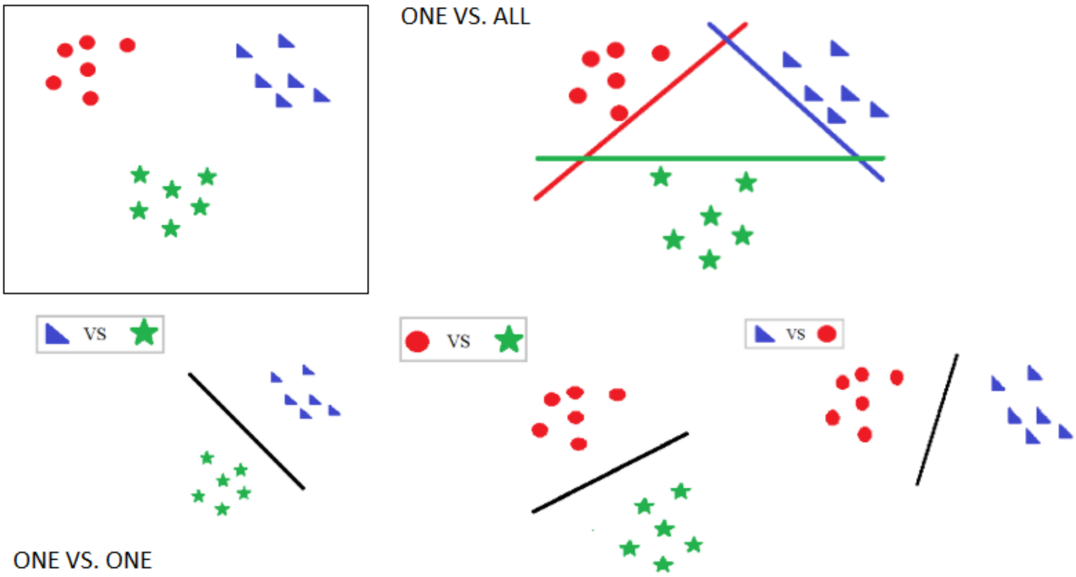
În partea stângă sunt prezentate drepte de decizie posibile pentru clasificarea celor două tipuri de obiecte. SVM-ul, exemplificat în partea dreaptă, alege hiperplanul care maximizează marginea dintre cele doua clase.

Pentru implementarea acestui algoritm vom folosi biblioteca ***ScikitLearn***. Aceasta este dezvoltată în Python, fiind integrată cu NumPy și pune la dispoziție o serie de algoritmi optimizați pentru probleme de clasificare, regresie și clusterizare.

Importarea modelului:

```
from sklearn import svm
```

Detalii de implementare:



Există două abordări pentru a clasifica datele aparținând mai multor clase:

1. **ONE VS ALL**: Sunt antrenați $num\_classes$ clasificatori, câte unul corespunzător fiecărei clase, care să o diferențieze pe aceasta de toate celelalte (toate celelalte exemple sunt privite ca aparținând aceleiași clase). Eticheta finală pentru un exemplu nou va fi dată de clasificatorul care a obținut scorul maxim.
2. **ONE VS ONE**: Sunt antrenați $\frac{num\_classes * (num\_classes - 1)}{2}$ clasificatori, câte unul corespunzător fiecarei perechi de câte două clase. Eticheta finală pentru un exemplu nou va fi cea care obține cele mai multe voturi pe baza acestor clasificatori.

➔ Implementarea din ScikitLearn are o abordare one-vs-one, adică pentru fiecare 2 clase este antrenat un clasificator binar care să diferențieze între acestea. Astfel, dacă avem un număr de clase egal cu $num\_classes$, vor fi antrenați $\frac{num\_classes * (num\_classes - 1)}{2}$ clasificatori.

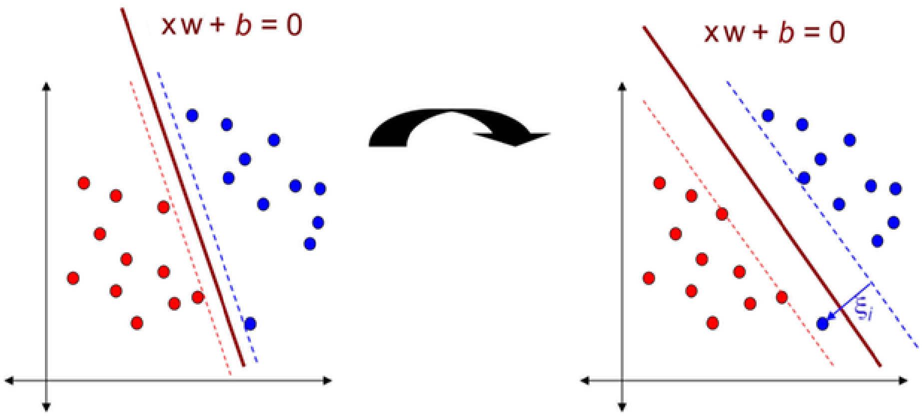
➔ La testare, clasa asignată fiecărui exemplu este cea care obține cele mai multe voturi pe baza acestor clasificatori.

1. Definirea modelului:

```
class sklearn.svm.SVC(C, kernel, gamma)
```

<u>Parametri:</u>

C (float, default = 1.0)



Influența parametrului C în alegerea marginii optime: în partea stângă este folosită abordarea hard margin, în care clasificatorul nu este dispus să clasifice greșit date de antrenare, iar în partea dreaptă este folosită abordarea soft margin. Variabila ξ_i sugerează cât de mult exemplul x_i are voie să depășească marginea.

$$\xi_i = max(0,\ 1\ -\ y_i(< x,\ w >\ +\ b))$$

- parametru de penalitate pentru eroare, sugerează cât de mult este dispus modelul să evite clasificarea greșită a exemplelor din setul de antrenare:
    - C mare - va fi ales un hiperplan cu o margine mai mică, dacă acesta are rezultate mai bune pe setul de antrenare (mai mulți vectori suport).

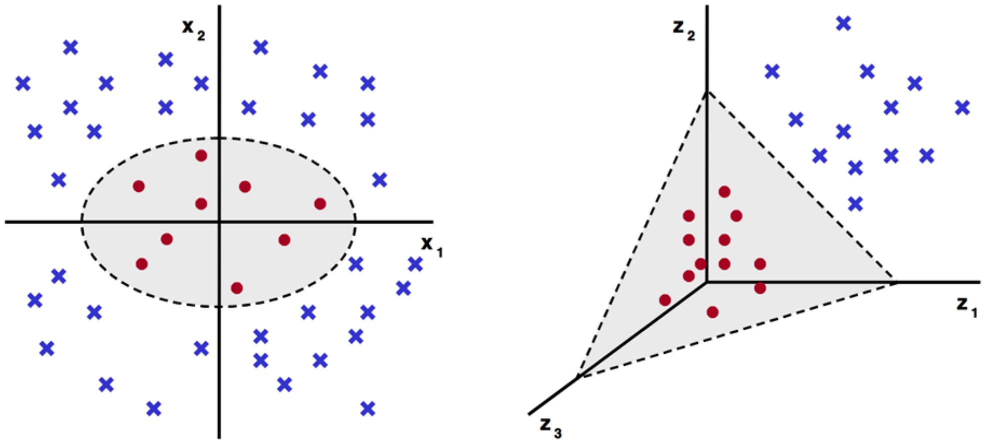
      Dacă C va fi ales prea mare, se poate ajunge la supra-învățare (overfitting).

    - C mic - va fi ales un hiperplan cu o margine mai mare, chiar dacă acesta duce la clasificarea greșită a unor puncte din setul de antrenare (mai puțini vectori suport).

    Dacă C va fi ales prea mic, modelul nu va fi capabil să învețe, ajungându-se la sub-învățare (underfitting).

kernel (string, default = ‘linear’)

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) = \left(x_1^2, \sqrt{2}x_1x_2, x_2^2\right)$$



Funcțiile kernel sunt folosite atunci când datele nu sunt liniar separabile. Acestea funcționează prin următorii doi pași:

1. Datele sunt scufundate într-un spațiu (Hilbert) cu mai multe dimensiuni
2. Relațiile liniare sunt căutate în acest spațiu

- tipul de kernel folosit: în cadrul laboratorului vom lucra cu ‘linear’ și ‘rbf’

Kernel linear:

$$K(u, v) = u^T v$$

Kernel RBF:

$$K(u, v) = exp(-\ gamma\ *\ ||u - v||^2)$$

gamma (float, default = ’auto’, având valoarea $\frac{1}{num\_features}$)

- coeficient pentru kernelul ‘rbf’
- dacă gamma = ‘scale’ va fi folosită valoarea $\frac{1}{num\_features * X.std()}$
- în versiunea 0.22 valoarea default ‘auto’ va fi schimbată cu ‘scale’

Sms Spam Classification

Această bază de date conține mesaje (sms) text spam/non-spam. Scopul nostru este să clasificăm dacă un mesaj este spam sau nu. Baza de date conține 3734 exemple de antrenare și 1840 exemple de testare. Raportul mesajelor non-spam:spam este de 6:1.

Exemple din setul de date:

spam URGENT! We are trying to contact you. Last weekends draw shows that you have won a £900 prize GUARANTEED. Call 09061701939. Claim code S89. Valid 12hrs only
ham Hi frnd, which is best way to avoid missunderstding wit our beloved one's?
ham Great escape. I fancy the bridge but needs her lager. See you tomo

Descărcați arhiva care conține datele de antrenare și testare [de aici.](#)

Exerciții

1. Descărcați arhiva ***[de aici](#)*** și observați cum funcționează modelul SVM.

2. Definiți funcția **normalize_data(train_data, test_data, type=None)** care primește ca parametri datele de antrenare, respectiv de testare și tipul de normalizare ({None, *'standard'*, *'l1'*, *'l2'*}) și întoarce aceste date normalizate.

3. Definiți clasa **BagOfWords** în al cărui constructor se inițializează vocabularul (un dicționar gol). În cadrul ei implementați metoda **build_vocabulary(self, data)** care primește ca parametru o listă de mesaje(listă de liste de strings) și construiește vocabularul pe baza acesteia. Cheile dicționarului sunt reprezentate de cuvintele din eseuri, iar valorile de id-urile unice atribuite cuvintelor. Pe lângă vocabularul pe care-l construiți, rețineți și o listă cu cuvintele în ordinea adăugării în vocabular.
Afișați dimensiunea vocabularul construit (9522).

   OBS. Vocabularul va fi construit doar pe baza datelor din setul de antrenare.

4. Definiți metoda **get_features(self, data)** care primește ca parametru o listă de mesaje de dimensiune $num\_samples$(listă de liste de strings) și returnează o matrice de dimensiune ($num\_samples$ x $dictionary\_length$) definită astfel:

$$features(sample\_idx,\ word\_idx)\ = numarul\ de\ aparitii\ al\ cuvantului\ cu\ id - ul\ word\_idx\ in\ documentul\ sample\_idx$$

5. Cu ajutorul funcțiilor definite anterior, obțineți reprezentările BOW pentru mulțimea de antrenare și testare, apoi normalizați-le folosind norma "L2".

6. Antrenați un SVM cu ***kernel linear*** care să clasifice mesaje în mesaje spam/non-spam. Pentru parametrul **C** setați valoarea **1**. Calculați acuratețea și F1-score pentru mulțimea de testare.
Afișați cele mai negative (spam) 10 cuvinte și cele mai pozitive (non-spam) 10 cuvinte.

   the first 10 negative words are ['Text' 'To' 'mobile' 'CALL' 'FREE' 'txt' '&' 'Call' 'Txt' 'STOP']
   the first 10 positive words are ['&lt#&gt' 'me' 'i' 'Going' 'him' 'Ok' 'I' 'Ill' 'my' 'Im']

Regresia Liniară. Regresia Ridge

1. Regresia Liniară

Dorim să găsim o funcție *g* astfel încât:

$$y_{hat} = g(X) = \Sigma_{i=1}{}^{i=n} x_i w_i + b$$

și care interpolează cel mai bine o mulțime de exemple $(X_1, y_1), (X_2, y_2), .., (X_n, y_n)$. Pentru a găsi această funcție, vom minimiza valoarea funcției **M**ean **S**quared **E**rror pe mulțimea de antrenare.

$$MSE(y, y_{hat}) = \Sigma_{i=1}{}^{i=n} (y_{hat_i} - y_i)^2$$

2. Regresia Ridge

Regresia Ridge adaugă o nouă "penalizare" funcției de cost, pe lângă faptul că diferența între etichetele *ground-truth* și etichetele *prezise* trebuie să fie minimă, dorim ca ponderile pe care le învățăm să fie mici. Pentru a forța ponderile să fie mici, vom adaugă la funcția de cost norma L_2 a ponderilor.

$$cost_{Ridge}(y, y_{hat}) = \Sigma_{i=1}{}^{i=n} (y_{hat_i} - y_i)^2 + \alpha \left|\left|W\right|\right|_2$$

Parametrul α controlează cât de mici să fie ponderile.

3. Regresia Lasso

Regresia Lasso adaugă norma L_1 a ponderilor la funcția de cost, creând o reprezentare *sparse* a ponderilor.

$$cost_{Lasso}(y, y_{hat}) = \Sigma_{i=1}{}^{i=n} (y_{hat_i} - y_i)^2 + \alpha \left|\left|W\right|\right|_1$$

În acest laborator vom folosi modelele implementate în biblioteca Scikit-Learn.

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
# definirea modelelor
linear_regression_model = LinearRegression()
ridge_regression_model = Ridge(alpha=1)
lasso_regression_model = Lasso(alpha=1)

# calcularea valorii MSE și MAE
from sklearn.metrics import mean_squared_error, mean_absolute_error
mse_value = mean_squared_error(y_true, y_pred)
mae_value = mean_absolute_error(y_true, y_pred)
```

Car Price Prediction

În continuare, vom lucra pe baza de date ***Car Price Prediction*** pentru a prezice prețul unei mașinii în funcție de caracteristicile ei.

Această bază de date este formată din 4879 exemple de antrenare. Neavând o mulțime separată de testare vom folosi tehnica de validare încrucișată (*cross-validation*) pentru a valida parametrii modelelor pe care le vom antrena.

În figura de mai jos, vedem 4 exemple din mulțime de antrenare.

Year	Kilometers_Driven	Fuel_Type	Transmission	Owner_Type	Mileage	Engine	Power	Seats	Price
2010	72000	CNG	Manual	First	26.6 km/kg	998 CC	58.16 bhp	5	1.75
2012	87000	Diesel	Manual	First	20.77 kmpl	1248 CC	88.76 bhp	7	6
2013	40670	Diesel	Automatic	Second	15.2 kmpl	1968 CC	140.8 bhp	5	17.74
2012	75000	LPG	Manual	First	21.1 km/kg	814 CC	55.2 bhp	5	2.35

După procesarea datelor (extragerea datelor din CVS și salvarea lor în format .npy) atributele au fost rearanjate în felul următor:

1. anul fabricației
2. numărul de kilometrii
3. mileage
4. motor
5. putere
6. numărul de locuri
7. numărul de proprietarii (valori între 1 și 4)

8-12. tipul de combustibil - fiind 5 tipuri de combustibil, acesta a fost recodat într-un one-hot vector de 5 componente.

13-14. tipul de transmisie - fiind 2 tipuri de transmisie, acesta a fost recodat într-un one-hot vector de 2 componente. 10 - „Manual”; 01 - ”Automatic”.

Descărcați arhiva care conține datele de antrenare [de aici.]()

Codul următor ne ajută să citim datele de antrenare:

```python
import numpy as np
from sklearn.utils import shuffle

# load training data
training_data = np.load('data/training_data.npy')
prices = np.load('data/prices.npy')
# print the first 4 samples
print('The first 4 samples are:\n ', training_data[:4])
print('The first 4 prices are:\n ', prices[:4])
# shuffle
training_data, prices = shuffle(training_data, prices, random_state=0)
```

Exerciții

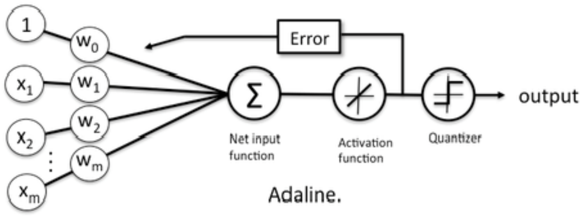
1. Definiți o metodă care primește doi parametrii, datele de antrenare și cele de testare și returnează datele normalizate. Folosiți o metodă de normalizare ***corespunzătoare*** pentru setul de date ***Car Price Prediction***.

2. Folosind mulțimea de antrenare din setul de date ***Car Price Prediction*** antrenați un *model de regresie liniară* folosind validarea încrucișată cu 3 fold-uri. Calculați valoarea medie a funcțiilor *MSE* și *MAE*.
   Nu uitați să normalizați datele folosind metoda definită anterior.

3. Folosind mulțimea de antrenare din setul de date ***Car Price Prediction*** antrenați un *model de regresie ridge* folosind validarea încrucișată cu 3 fold-uri. Calculați valoarea medie a funcțiilor *MSE* și *MAE*. Verificați care valoare a lui α, $\alpha \in \{1, 10, 100, 1000\}$obține o performanță mai bună.
   Nu uitați să normalizați datele folosind metoda definită anterior.

4. Folosind cel mai performant *alpha* de la punctul anterior, antrenați un *model de regresie ridge* pe întreaga mulțime de antrenare, afișați coeficienți și bias-ul regresiei. Care este *cel mai semnificativ* atribut? Care este al doilea *cel mai semnificativ atribut*? Care este *cel mai puțin semnificativ* atribut?
   Nu uitați să normalizați datele folosind metoda definită anterior.

Perceptronul și rețele de perceptroni



Structura unui perceptron cu *m* ponderi. Functia de predictie a perceptronului este $y_{hat} = sign(\sum_{i=0}^{i=m} x_i * w_i)$.

1. Perceptronul

Perceptronul este un clasificator liniar. Predictia clasificatorului pentru exemplul $X = \{x_1, x_2, \dots, x_n\}$ este $y_{hat} = f(\sum_{i=1}^{i=n} x_i * w_i + b)$, unde $W = \{w_1, w_2, \dots, w_n\}$ si $b = w_0$ sunt ponderile, respectiv bias-ul perceptronului, iar f este functia de transfer (numita si functie de activare). Putem inlocui suma din calcularea lui y_{hat} cu produsului dintre vectorul datelor de intrare X si matricea ponderilor$W$, rezultatand $y_{hat} = f(X \cdot W + b)$.

2. Algoritmul Widrow-Hoff.

Algoritmul Widrow-Hoff, numit si *metoda celor mai mici patrate* (*Least mean squares*), este un algoritm de optimizare a erorii perceptronului pe baza metodei coborarii pe gradient tinand cont doar de eroare de la exemplul curent.

Regula de actualizare foloseste derivata partiala a functiei de pierdere, in functie de ponderi si bias. In continuare vom calcula detaliat derivatele partiale ale functiei de pierdere. Functia de activare a perceptronului din algoritmul Widrow-Hoff este *identitatea* ($f(x) = x$).

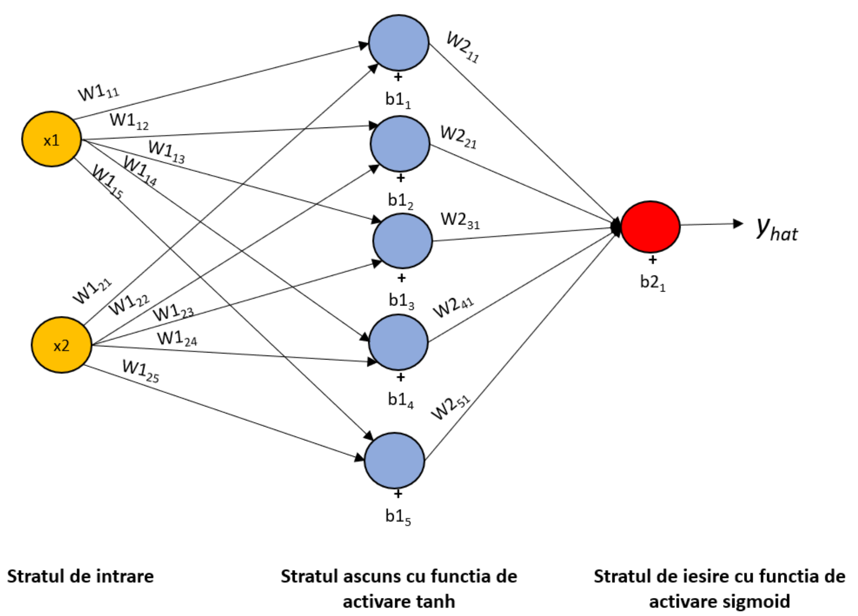
$$loss = \frac{(y_{hat} - y)^2}{2}, unde\ y_{hat} = X \cdot W + b, iar\ y\ este\ eticheta\ lui\ X$$

∂W	∂b
$\frac{\partial loss}{\partial W} = \frac{\partial \frac{(y_{hat} - y)^2}{2}}{\partial W}$	$\frac{\partial loss}{\partial b} = \frac{\partial \frac{(y_{hat} - y)^2}{2}}{\partial b}$
$\frac{\partial loss}{\partial W} = \frac{\partial \frac{(x \cdot W + b - y)^2}{2}}{\partial W}$	$\frac{\partial loss}{\partial b} = \frac{\partial \frac{(x \cdot W + b - y)^2}{2}}{\partial b}$
$\frac{\partial loss}{\partial W} = \frac{2 \cdot (x \cdot W + b - y) \cdot \frac{\partial (x \cdot W + b - y)}{\partial W}}{2}$	$\frac{\partial loss}{\partial b} = \frac{2 \cdot (x \cdot W + b - y) \cdot \frac{\partial (x \cdot W + b - y)}{\partial b}}{2}$
$\frac{\partial loss}{\partial W} = (x \cdot W + b - y) \cdot x$	$\frac{\partial loss}{\partial b} = (x \cdot W + b - y) \cdot 1$
$\frac{\partial loss}{\partial W} = (y_{hat} - y) \cdot x$	$\frac{\partial loss}{\partial b} = (y_{hat} - y)$

Algoritmul Widrow-Hoff.

1. $X = \{x_0, x_1, \dots, x_{T-1}\}, X \in R^{TxN} - datele\ de\ intrare, Y = \{y_0, y_1, \dots, y_{T-1}\} - etichetele$
2. $W = \{w_0, w_1, \dots, w_{N-1}\} = 0\ ;\ b = 0 \quad //\ initializeaza\ ponderile\ cu\ un\ vector\ de\ 0 - uri$
3. $pentru\ e = 0{:}\ E - 1\ executa{:} \quad //\ pentru\ fiecare\ epoca$
   a. $amesteca\ datele\ de\ antrenare$
   b. $pentru\ t = 0{:}\ T - 1\ executa{:} \quad //\ pentru\ fiecare\ exemplu\ x_t\ din\ X$
      i. $y_{hat} = x_t \cdot W + b \quad //\ calculeaza\ predictia$
      ii. $loss = \frac{(y_{hat} - y_t)^2}{2} \quad //\ calculeaza\ eroarea\ pentru\ exemplul\ x_t$
      iii. $W = W - \eta(y_{hat} - y_t)x_t \quad //\ actualizeaza\ ponderile\ folosind\ \frac{\partial loss}{\partial W}$
      iv. $b = b - \eta(y_{hat} - y_t) \quad //\ actualizeaza\ bias - ul\ folosind\ \frac{\partial loss}{\partial b}$

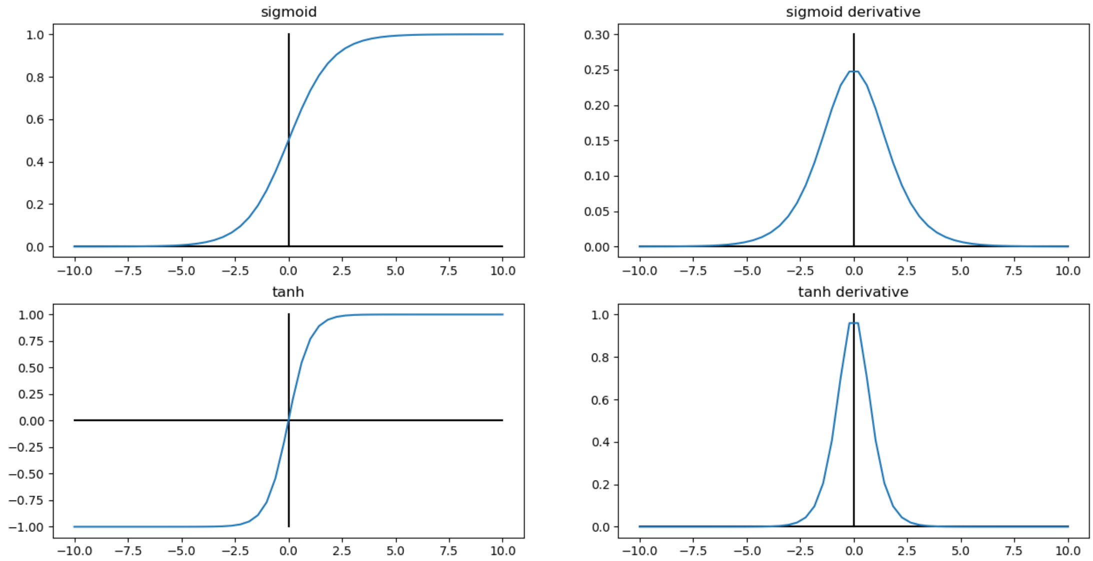
3. Retele feedforward de perceptroni



O retea neuronala cu 5 perceptronii pe stratul ascuns si un perceptron pe stratul de iesire.

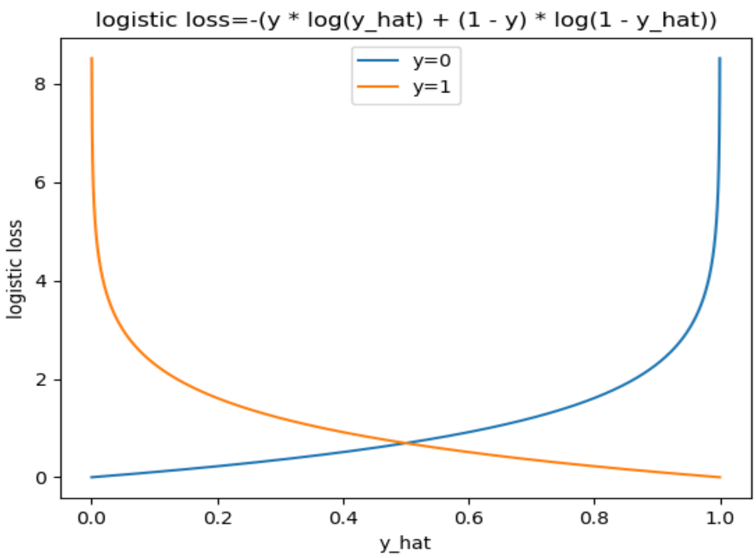
Retelele neurale feedforward sunt retele de perceptroni grupati pe straturi, in care propagarea informatiei se realizeaza numai dinspre intrare spre iesire (de la stanga la dreapta). Retelele feedforward sunt multistrat, continand mai multe straturi de perceptroni. Perceptronii de pe primul strat sunt singurii care primesc date de intrare din exterior. Perceptronii de pe celelalte straturi (numite *straturi ascunse (hidden layers)*), primesc ca date de intrare rezultatul stratului anterior. Ultimul strat din retea se numeste *strat de iesire (output layer)*.

In cadrul laboratorului vom antrena o retea cu un strat ascuns cu ***num_hidden_neurons*** neuroni si functia de activare ***tanh*** si un neuron pe stratul de iesire cu functie de activare ***logistic*** (sigmoid) pentru rezolvarea problemei **XOR**. Predictia retelei pentru un exemplu X este $y_{hat} = sigmoid(tanh(X \cdot W_1 + b_1) \cdot W_2 + b_2)$.



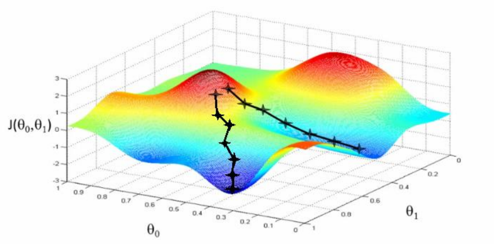
Stânga -sus: graficul functiei sigmoid; *Dreapta-jos*: graficul functiei sigmoid derivat. *Stânga Jos*:graficul funcției tanh; *Dreapta-jos*: graficul functiei tanh derivat.

Functia de pierdere pe care o vom folosi pentru antrenarea retelei este: $logistic\_loss(y_{hat}, y) = -(y * log(y_{hat}) + (1 - y) * log(1 - y_{hat}))$, unde y_{hat} este predicția rețelei pentru exemplul X, iar y este eticheta binara (0 sau 1) a lui X



Linia portocalie: valoarea functiei *logistic loss*, cand y=1, iar *y_hat* variaza intre (0,1). Observam ca cu cat ne apropiem de 1 (pe axa Ox) valoarea functiei scade. Se observa ca daca y=1, valoarea functiei este data doar de produsul din partea stanga (partea dreapta inmultindu-se cu 0).
Linia albastra: valoarea functiei *logistic loss*, cand y=0, iar *y_hat* variaza intre (0,1). Observam ca cu cat ne indepartam de 0 (pe axa Ox) valoarea functiei creste. Se observa ca daca y=0, valoarea functiei este data doar de produsul din partea dreapta (partea stanga inmultindu-se cu 0).

4. Antrenarea retelelor de perceptroni cu algoritmul coborarii pe gradient



Observam ca in functie de initializarea ponderilor putem ajunge in minime locale diferite.

Algoritmul coborarii pe gradient se bazeaza pe derivata de ordinul 1, pentru a gasi minimul functiei de pierdere. Pentru a gasi un minim local al functiei de pierdere, vom actualiza ponderile retelei proportional cu negativul gradientului functiei la pasul curent.

In continuare vom detalia implementarea (pseudo-cod) algoritmului de coborare pe gradient pentru reteaua descrisa anterior.

Pasii algoritmului sunt:

1) Initializare ponderilor - ponderile si bias-ul retelei se initializeaza aleator cu valori mici aproape de 0 sau cu valoare 0.

```
W_1 = random((2, num_hidden_neurons), miu, sigma)
# generam aleator matricea ponderilor stratului ascuns (2 - dimensiunea datelor de intrare, num_hidden_neurons - numarul neuronilor de pe stratul ascuns), cu media miu  si deviatia standard sigma.
b_1 = zeros(num_hidden_neurons) # initializam bias-ul cu 0
W_2 = random((num_hidden_neurons, 1), miu, sigma)
# generam aleator matricea ponderilor stratului de iesire (num_hidden_neurons - numarul neuronilor de pe stratul ascuns, 1 - un neuron pe stratul de iesire), cu media miu  si deviatia standard sigma.
b_2 = zeros(1) # initializam bias-ul cu 0
```

2) Pasul **forward -** Vom defini o metoda forward care calculeaza predictia retelei folosind ponderile actuale si datele de intrare date ca parametri, apoi vom calcula pentru fiecare strat valoarea lui ***z*** (***z*** = inmultirea datelor de intrare cu ponderile si adunarea bias-ului) si valoarea lui ***a*** (***a*** = aplicarea functiei de activare lui ***z***, ($a = f(z)$)).

```
forward(X, W_1, b_1, W_2, b_2)
# X - datele de intrare,  W_1, b_1, W_2 si b_2 sunt ponderile
```

```
retelei
z_1 =  X * W_1 + b_1
a_1 = tanh(z_1)
z_2 = a_1 * W_2 + b_2
a_2 = sigmoid(z_2)
return z_1, a_1, z_2, a_2 # vom returna toate elementele
calculate
```

3) Calculam valoarea functiei de eroare (logistic loss) si acuratetea.

```
z_1, a_1, z_2, a_2 = forward(X, W_1, b_1, W_2, b_2)
loss = (-y .* log(a_2) - (1 - y) .* log(1 - a_2)).mean()
accuracy = (round(a_2) == y).mean()
```

functia	***derivata***	***Derivata functiei compuse***
$sigmoid(x) = \frac{1}{1+e^{-x}}$	$sigmoid(x) * (1 - sigmoid(x))$	$sigmoid(u(x)) * (1 - sigmoid(u(x))) * u(x)'$
$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$	$1 - tanh(x)^2$	$(1 - tanh(u(x))^2) * u(x)'$
x	1	$-$
$c * x$	c	$-$
$ln\, x$	$\frac{1}{x}$	$\frac{u(x)'}{u(x)}$
x^n	$n * x^{n-1}$	$n * u(x)^{n-1} * u(x)'$

Derivatele functiilor folosite in laborator.

4) Pasul **backward** - vom defini o metoda backward care calculeaza derivata functiei de eroare pe directiile ponderilor, respectiv a fiecarui bias. Vom incepe calculul cu derivata functiei de pierdere pe directia ***z_2*** folosind regula de inlantuire (chain-rule) a derivatelor.

$$\frac{\partial loss}{\partial z_2} = \frac{\partial loss}{\partial a_2} * \frac{\partial a_2}{\partial z_2} \mid aplicam\ regula\ de\ inlantuire$$

$Stim\ ca\ \ a_2 = sigmoid(z_2)\ , folosind\ derivata\ functiei\ sigmoid\ \ rezulta:$

$$\frac{\partial a_2}{\partial z_2} = \frac{sigmoid(z_2)}{\partial z_2} = sigmoid(z_2) * (1 - sigmoid(z_2)) = a_2 * (1 - a_2)$$

$$\frac{\partial loss}{\partial a_2} = \frac{\partial(-y * log(a_2) - (1-y) * log(1-a_2))}{\partial a_2}$$

$$\frac{\partial loss}{\partial a_2} = \left(\frac{-y}{a_2} + \frac{1-y}{1-a_2}\right)$$

$$\frac{\partial loss}{\partial a_2} = \frac{-y + a_2 * y + a_2 - a_2 * y}{a_2 * (1-a_2)}$$

$$\frac{\partial loss}{\partial z_2} = \frac{\partial loss}{\partial a_2} * \frac{\partial a_2}{\partial z_2}$$

$$\frac{\partial loss}{\partial z_2} = \frac{-y + a_2 * y + a_2 - a_2 * y}{a_2 * (1-a_2)} * a_2 * (1 - a_2)$$

$$\frac{\partial loss}{\partial z_2} = a_2 - y$$

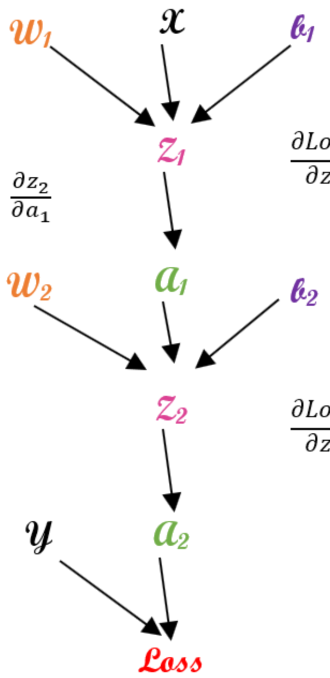
$$\frac{\partial Loss}{\partial W_1} = \frac{\partial Loss}{\partial z_1} * \frac{\partial z_1}{\partial W_1} \qquad \frac{\partial Loss}{\partial b_1} = \frac{\partial Loss}{\partial z_1} * \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial Loss}{\partial z_1} = \frac{\partial Loss}{\partial a_1} * \frac{\partial a_1}{\partial z_1}$$

$$\frac{\partial Loss}{\partial a_1} = \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial a_1}$$

$$\frac{\partial Loss}{\partial W_2} = \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial W_2} \qquad \frac{\partial Loss}{\partial b_2} = \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial b_2}$$

$$\frac{\partial Loss}{\partial z_2} = \frac{\partial Loss}{\partial a_2} * \frac{\partial a_2}{\partial z_2}$$



Calcularea derivatele partiale pe directiile ponderilor si a fiecarui bias folosind regula de inlantuire.

```
backward(a_1, a_2, z_1, W_2, X, Y, num_samples)
dz_2 = a_2 - y # derivata functiei de pierdere (logistic loss) in functie de z
dw_2 = (a_1.T * d_z2) / num_samples
# der(L/w_2) = der(L/z_2) * der(dz_2/w_2) = dz_2 * der((a_1 * W_2 + b_2)/ W_2)
db_2 = sum(dz_2) / num_samples
# der(L/b_2) = der(L/z_2) * der(z_2/b_2) = dz_2 * der((a_1 * W_2 + b_2)/ b_2)
# primul strat
da_1 = dz_2 * W_2.T
```

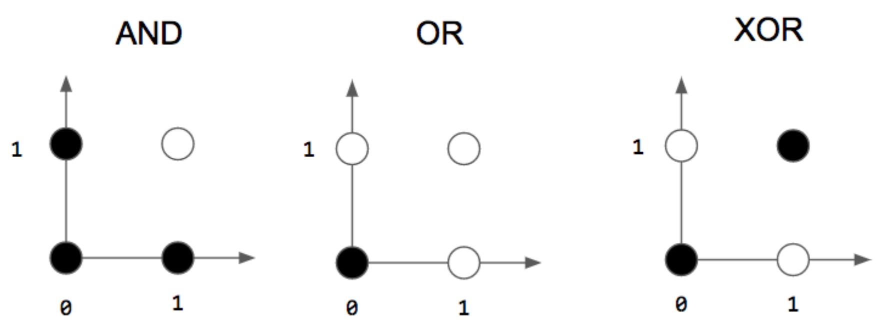
```
# der(L/a_1) = der(L/z_2) * der(z_2/a_1) = dz_2 * der((a_1 * W_2 + b_2)/ a_1)
dz_1 = da_1 .* tanh_derivative(z_1)
# der(L/z_1) = der(L/a_1) * der(a_1/z1) = da_1 .* der((tanh(z_1))/ z_1)
dw_1 = X.T * dz_1 / num_samples
# der(L/w_1) = der(L/z_1) * der(z_1/w_1) = dz_1 * der((X * W_1 + b_1)/ W_1)
db_1 = sum(dz_1) / num_samples
# der(L/b_1) = der(L/z_1) * der(z_1/b_1) = dz_1 * der((X * W_1 + b_1)/ b_1)
return dw_1, db_1, dw_2, db_2
```

5) Actualizarea ponderilor - ponderile se actualizeaza proportional cu negativul mediei derivatelor din batch (mini-batch).

```
W_1 -= lr * dw_1 # lr - rata de invatare (learning rate)
b_1 -= lr * db_1
W_2 -= lr * dw_2
b_2 -= lr * db_2
```

6) Pentru a antrena o retea neuronala cu ajutorul algoritmului coborarii pe gradient trebuie sa:
   a) Stabilim numarul de epoci
   b) Stabilim rata de invatare
   c) Sa initiliazam ponderile (pasul 1)
   d) Sa amestecam datele la fiecare epoca
   e) Sa luam un subset din multimea (sau toata multimea) de antrenare si sa executam pasii 2, 3, 4, 5 pana la convergenta.

Exercitii



1. Se da urmatoare multime de antrenare X =[[0, 0], [0, 1], [1, 0], [1, 1]], y = [-1, 1, 1, 1]. Sa se gaseasca o dreapta care separa perfect multimea de antrenare.

2. Antrenati un Perceptron cu algoritmul Widrow-Hoff pe multimea de antrenare de la exercitiul anterior timp de 70 epoci cu rata de invatare 0.1. Care este acuratetea pe multimea de antrenare? Apelati functia *plot_decision_boundary* la fiecare pas al algoritmului pentru a afisa dreapta de decizie.

```
import matplotlib.pyplot as plt

def compute_y(x, W, bias):
    # dreapta de decizie
    # [x, y] * [W[0], W[1]] + b = 0
    return (-x * W[0] - bias) / (W[1] + 1e-10)

def plot_decision_boundary(X, y , W, b, current_x, current_y):
    x1 = -0.5
    y1 = compute_y(x1, W, b)
    x2 = 0.5
    y2 = compute_y(x2, W, b)
    # sterge continutul ferestrei
    plt.clf()
    # ploteaza multimea de antrenare
    color = 'r'
    if(current_y == -1):
        color = 'b'
    plt.ylim((-1, 2))
    plt.xlim((-1, 2))
    plt.plot(X[y == -1, 0], X[y == -1, 1], 'b+')
    plt.plot(X[y == 1, 0], X[y == 1, 1], 'r+')
    # ploteaza exemplul curent
    plt.plot(current_x[0], current_x[1], color+'s')
    # afisarea dreptei de decizie
    plt.plot([x1, x2] ,[y1, y2], 'black')
    plt.show(block=False)
    plt.pause(0.3)
```

3. Antrenati un Perceptron cu algoritmul Widrow-Hoff pe multimea de antrenare X =[[0, 0], [0, 1], [1, 0], [1, 1]], y = [-1, 1, 1, -1]. Care este acuratetea pe multimea de antrenare? Apelati functia *plot_decision_boundary* la fiecare pas al algoritmului pentru a afisa dreapta de decizie.
4. Antrenati o retea neuronala pentru rezolvarea problemei XOR cu arhitectura retelei descrise in **3**, si algoritmul coborarii pe gradient descris in **4**, folosind 70 epoci, rata de invatare 0.5, media si deviatia standard pentru initializarea ponderilor 0, respectiv 1, si 5 neuroni pe stratul ascuns. Afisati valoarea erorii si a acuratetii la fiecare epoca. Apelati functia *plot_decision* la fiecare pas al algoritmului pentru a afisa functia de decizie.

```
def compute_y(x, W, bias):
    # dreapta de decizie
    # [x, y] * [W[0], W[1]] + b = 0
    return (-x*W[0] - bias) / (W[1] + 1e-10)
```

```python
def plot_decision(X_, W_1, W_2, b_1, b_2):
    # sterge continutul ferestrei
    plt.clf()
    # ploteaza multimea de antrenare
    plt.ylim((-0.5, 1.5))
    plt.xlim((-0.5, 1.5))
    xx = np.random.normal(0, 1, (100000))
    yy = np.random.normal(0, 1, (100000))
    X = np.array([xx, yy]).transpose()
    X = np.concatenate((X, X_))
    _, _, _, output = forward(X, W_1, b_1, W_2, b_2)
    y = np.squeeze(np.round(output))
    plt.plot(X[y == 0, 0], X[y == 0, 1], 'b+')
    plt.plot(X[y == 1, 0], X[y == 1, 1], 'r+')
    plt.show(block=False)
    plt.pause(0.1)
```