Prácticas de Aprendizaje Automático

Clase 2: Ejercicios/ejemplos de repaso

Pablo Mesejo y Salvador García
Universidad de Granada
Departamento de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE GRANADA

Índice

1. Ejercicios/ejemplos de repaso de Python

2. Ejercicios/ejemplos de repaso de NumPy y Matplotlib

Ejercicios/ejemplos de repaso de Python

Repaso Python (1)

```
def funcion1(arr,value=3):
   return([x*2 for x in arr if x < value//2])


funcion1([3,6,8,10,1,2,1],5)
```

Repaso Python (1)

```
def funcion1(arr,value=3):
   return([x*2 for x in arr if x < value//2])


funcion1([3,6,8,10,1,2,1],5)
```

```
[2,2]
```

Repaso Python (2)

¿Podríais decir la función que se corresponde con este código Python?

```
E(u,v) = (u**3*np.exp(v-2)-4*v**3*np.exp(-u))**2
```

Repaso Python (2)

¿Podríais decir la función que se corresponde con este código Python?

```
E(u,v) = (u**3*np.exp(v-2)-4*v**3*np.exp(-u))**2
```

$$E(u,v) = (u^3 e^{(v-2)} - 4v^3 e^{-u})^2$$

Repaso Python (3)

¿Alguien sabe qué hace esta función?

```
def someGreatFunction(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return someGreatFunction(left) + middle + someGreatFunction(right)

print(someGreatFunction([3,6,8,10,1,2,1]))
```

Ejemplo extraído de Andrej Karpathy: http://cs231n.github.io/python-numpy-tutorial/

Repaso Python (3)

```
def QuickSort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return QuickSort(left)+middle+QuickSort(right)

print(QuickSort([3,6,8,10,1,2,1]))
```

```
pivot: 10
left:  [3, 6, 8, 1, 2, 1]
middle:  [10]
right:  []
---------------------------------------
pivot: 1
left:  []
middle:  [1, 1]
right:  [3, 6, 8, 2]
---------------------------------------
pivot: 8
left:  [3, 6, 2]
middle:  [8]
right:  []
---------------------------------------
pivot: 6
left:  [3, 2]
middle:  [6]
right:  []
---------------------------------------
pivot: 2
left:  []
middle:  [2]
right:  [3]
---------------------------------------
[1, 1, 2, 3, 6, 8, 10]
```

Ejemplo extraído de Andrej Karpathy: http://cs231n.github.io/python-numpy-tutorial/

Repaso Python (4)

¿Qué imprime este código?

```
def fun1(b):
    b.append(1)
    b = 'New Value'
    print('Dentro de fun1: ', b)

a = [0]
fun1(a)
print('Despues de fun1: ', a)
```

Repaso Python (4)

¿Qué imprime este código?

```
def fun1(b):
    b.append(1)
    b = 'New Value'
    print('Dentro de fun1: ', b)

a = [0]
fun1(a)
print('Despues de fun1: ', a)
```

Cuando se llama a **fun1**, **b** y **a** apuntan al mismo valor ([0])

Cuando hacemos **b.append(1)** → [0] se convierte en [0, 1]

Cuando hacemos **b = 'New Value'** → ahora **b** apunta a una nueva lista en memoria que contiene 'New Value'. Pero **a** apunta todavía a la lista [0, 1]

```
Dentro de fun1:  New Value
Despues de fun1:  [0, 1]
```

Repaso Python (4)

¿Y si queremos que el valor modificado se vea fuera (es decir, que cambie el valor de a)?

```
def fun1(b):
    b.append(1)
    b = 'New Value'
    print('Dentro de fun1: ', b)

a = [0]
fun1(a)
print('Despues de fun1: ', a)
```

Repaso Python (4)

¿Y si queremos que el valor modificado se vea fuera (es decir, que cambie el valor de a)?

```
def fun1(b):
    b.append(1)
    b = 'New Value'
    print('Dentro de fun1: ', b)
    return b

a = [0]
a = fun1(a)
print('Despues de fun1: ', a)
```

Introducimos un **`return`**

Reasignamos la variable a la salida de la función

```
Dentro de fun1:  New Value
Despues de fun1:  New Value
```

Repaso Python (5)

¿Qué contiene `tupla` tras la última asignación?

```
In [40]: tupla = (5, 't1', True, 0.5)

In [41]: tupla[2]
Out[41]: True

In [42]: tupla[2] = False
```

Repaso Python (5)

¿Qué contiene `tupla` tras la última asignación?

```
In [40]: tupla = (5, 't1', True, 0.5)

In [41]: tupla[2]
Out[41]: True

In [42]: tupla[2] = False

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-42-1f9f497d5a2b> in <module>
----> 1 tupla[2] = False

TypeError: 'tuple' object does not support item assignment
```

Repaso Python (6)

¿Qué ocurre al ejecutar este código?

```
In [1]: x = 1.0
   ...: y = 2.0
   ...: print = (x)
   ...: print(x)
```

Repaso Python (6)

¿Qué ocurre al ejecutar este código?

```
In [1]: x = 1.0
   ...: y = 2.0
   ...: print = (x)
   ...: print(x)
Traceback (most recent call last):

  File "C:\Users\pmese\AppData\Local\Temp/
ipykernel_18956/3544572640.py", line 4, in <module>
    print(x)

TypeError: 'float' object is not callable
```

Repaso Python (7)

Y si ahora intentamos imprimir algo, ¿qué ocurre?

```
In [1]: x = 1.0
   ...: y = 2.0
   ...: print = (x)
   ...: print(x)
Traceback (most recent call last):

  File "C:\Users\pmese\AppData\Local\Temp/
ipykernel_18956/3544572640.py", line 4, in <module>
    print(x)

TypeError: 'float' object is not callable


In [2]: print("HOLA, ME LLAMO PABLO")
```

Repaso Python (7)

Y si ahora intentamos imprimir algo, ¿qué ocurre?

```
In [1]: x = 1.0
   ...: y = 2.0
   ...: print = (x)
   ...: print(x)
Traceback (most recent call last):

  File "C:\Users\pmese\AppData\Local\Temp/
ipykernel_18956/3544572640.py", line 4, in <module>
    print(x)

TypeError: 'float' object is not callable


In [2]: print("HOLA, ME LLAMO PABLO")
Traceback (most recent call last):

  File "C:\Users\pmese\AppData\Local\Temp/
ipykernel_18956/3788885797.py", line 1, in <module>
    print("HOLA, ME LLAMO PABLO")

TypeError: 'float' object is not callable
```

¿Cómo podemos resolver este problema?

Hemos creado una variable global (**print = (x)**) que enmascara a la *built-in function*. Si la eliminamos, Python encontrará el built-in de nuevo.

del print

Repaso Python (8)

¿Qué imprime el siguiente código?

```
funcion_compuesta = lambda x, func: x + func(x)
funcion_compuesta(3, x ** 2)
```

Repaso Python (8)

¿Qué imprime el siguiente código?

```
funcion_compuesta = lambda x, func: x + func(x)
funcion_compuesta(3, x ** 2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-929b9e7d9ae0> in <module>
      1 funcion_compuesta = lambda x, func: x + func(x)
----> 2 funcion_compuesta(3, x ** 2)

NameError: name 'x' is not defined
```

Faltaría definir una *lambda function* que aporte esa 'x'

```
funcion_compuesta = lambda x, func: x + func(x)
funcion_compuesta(3, lambda x: x ** 2)
```

```
12
```

Repaso Python (y 9)

¿Qué imprime el siguiente código?

```
import numpy as np
y1 = np.array
y2 = y1([0,0])
print(y2)
```

Repaso Python (y 9)

¿Qué imprime el siguiente código?

```
import numpy as np
y1 = np.array
y2 = y1([0,0])
print(y2)
```

```
[0 0]
```

Y si ahora hacemos lo siguiente, ¿qué se imprime?

```
y1 + 1.0
```

Repaso Python (y 9)

¿Qué imprime el siguiente código?

```
import numpy as np
y1 = np.array
y2 = y1([0,0])
print(y2)
```

```
[0 0]
```

Ocurre algo parecido a lo que pasaba con **`print = (x)`**. **`y1`** se convierte, de facto, en **`np.array()`**. Da error porque estamos intentando hacer **`np.array + 1.0`** (sumar una función y un float). Deberíamos haber hecho **`y1 = np.array([0,0])`**.

Y si ahora hacemos lo siguiente, ¿qué se imprime?

```
y1 + 1.0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-ee169f520a78> in <cell line: 1>()
----> 1 y1 + 1.0

TypeError: unsupported operand type(s) for +: 'builtin_function_or_method' and 'float'
```

Ejercicios/ejemplos de repaso de NumPy y Matplotlib

Repaso NumPy (1)

¿Qué contiene X?

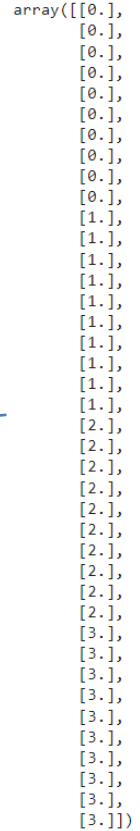
```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
```

Nota1: np.concatenate() concatena por defecto con respecto a axis=0 (es decir, por filas).

Repaso NumPy (1)

¿Qué contiene X?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
```



```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.]])
```

Nota1: np.concatenate() concatena por defecto con respecto a axis=0 (es decir, por filas).

Repaso NumPy (1)

¿Qué contiene X?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))
```

```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [2.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.],
       [3.]])
```

Nota1: np.concatenate() concatena por defecto con respecto a axis=0 (es decir, por filas).

Nota2: np.reshape() opera, por defecto, con respecto al axis=1. Es decir, rellenando los valores columna a columna!

Repaso NumPy (1)

¿Qué contiene X?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))
```

Concatena por defecto con respecto a axis=0. Aquí obtenemos un vector columna de 40 elementos

```
Out[2]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

El reshape se hace, por defecto, con respecto al axis=1 (**`order='C'`**)!
Es decir, rellenando los valores columna a columna!
Si os interesa hacerlo en el axis=0 (**`order='F'`**).

Si hiciésemos

```
X = np.reshape(X,(10,4),order='F')
```

```
array([[0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.],
       [0., 1., 2., 3.]])
```

Repaso NumPy (2)

¿Qué contiene y?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))

y = np.sum(X,axis=1)
```

```
Out[2]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

Nota: `np.sum(X,axis=1)` calcula la suma por filas. El concepto de "recorrer un eje" puede ser confuso. P.ej. sumar "por columnas" significa "recorrer la matriz por filas" (axis-0).
https://numpy.org/doc/1.13/glossary.html

Repaso NumPy (2)

¿Qué contiene y?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))

y = np.sum(X,axis=1)
```

```
Out[2]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

```
array([ 0.,  0.,  2.,  4.,  4.,  8.,  8., 10., 12., 12.])
```

Suma por filas! Es decir, recorriendo las columnas!

Repaso NumPy (3)

¿Qué contiene X_class?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))

y = np.sum(X,axis=1)

clases = np.unique(y)
X_class = [X[y==c_i] for c_i in clases]
```

```
Out[2]:
array([[0., 0., 0., 0.],   <- X[0]
       [0., 0., 0., 0.],   <- X[1]
       [0., 0., 1., 1.],   <- X[2]
       [1., 1., 1., 1.],   ...
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

```
array([ 0.,  0.,  2.,  4.,  4.,  8.,  8., 10., 12., 12.])
```

Repaso NumPy (3)

¿Qué contiene X_class?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))

y = np.sum(X,axis=1)

clases = np.unique(y)
X_class = [X[y==c_i] for c_i in clases]
```

```
Out[2]:
array([[0., 0., 0., 0.],   <- X[0]
       [0., 0., 0., 0.],   <- X[1]
       [0., 0., 1., 1.],   <- X[2]
       [1., 1., 1., 1.],   ...
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

```
array([ 0.,  0.,  2.,  4.,  4.,  8.,  8., 10., 12., 12.])
```

```
for c_i in clases:
    print(c_i)
0.0
2.0
4.0
8.0
10.0
12.0
```

Devuélveme, para cada **clase** (i.e., valor único en **y**), las posiciones correspondientes en **X**.

Repaso NumPy (3)

¿Qué contiene X_class?

```
import numpy as np

X1 = np.zeros((10,1))
X2 = np.ones((10,1))
X3 = np.ones((10,1))*2
X4 = np.ones((10,1))*3

X = np.concatenate((X1,X2,X3,X4))
X = np.reshape(X,(10,4))

y = np.sum(X,axis=1)

clases = np.unique(y)
X_class = [X[y==c_i] for c_i in clases]
```

X

```
Out[2]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

y

```
array([ 0.,  0.,  2.,  4.,  4.,  8.,  8., 10., 12., 12.])
```

```
Out[6]:
[array([[0., 0., 0., 0.],
        [0., 0., 0., 0.]]), array([[0., 0., 1., 1.]]), array([[1., 1., 1., 1.],
        [1., 1., 1., 1.]]), array([[2., 2., 2., 2.],
        [2., 2., 2., 2.]]), array([[2., 2., 3., 3.]]), array([[3., 3., 3., 3.],
        [3., 3., 3., 3.]])]
```

```
In [7]: X_class[0]
Out[7]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

y == 0.0

```
In [8]: X_class[1]
Out[8]: array([[0., 0., 1., 1.]])
```

y == 2.0

Repaso NumPy (4)

¿Qué hacen estas 4 líneas de código?

```
Z = np.arange(10)
v = np.random.uniform(0,10)
index = (np.abs(Z-v)).argmin()
print(Z[index])
```

Repaso NumPy (4)

¿Qué hacen estas 4 líneas de código?

```
Z = np.arange(10)
v = np.random.uniform(0,10)
index = (np.abs(Z-v)).argmin()
print(Z[index])
```

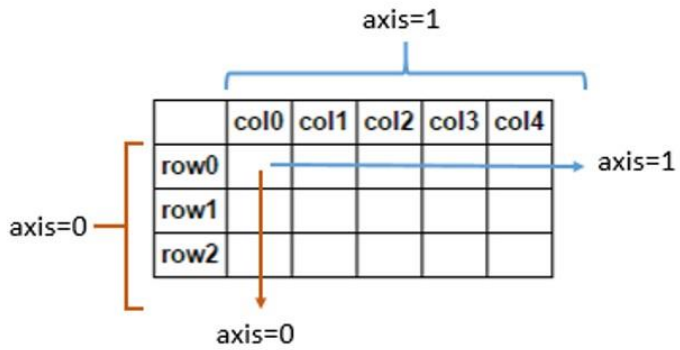
```
In [23]: Z
Out[23]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [24]: v
Out[24]: 4.1970769994524515

In [25]: index = (np.abs(Z-v)).argmin()
    ...: print(Z[index])
4

In [26]: index
Out[26]: 4
```

Dado un array Z, y un valor v, ¿cuál es el valor de Z más próximo a v?

Repaso NumPy (5)

¿Cómo es la matriz que imprimen de salida estas tres líneas de código?

```
a = np.arange(10).reshape(2,-1)
b = np.repeat(1, 10).reshape(2,-1)
np.concatenate([a, b], axis=0)
```



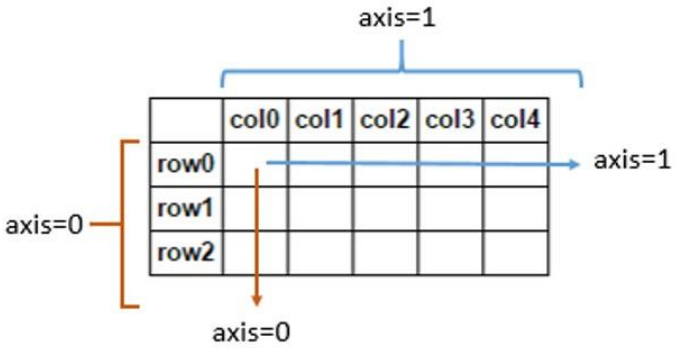
Nota1: np.concatenate() concatena por defecto con respecto a axis=0 (es decir, por filas).
Nota2: np.reshape() opera, por defecto, con respecto al axis=1. Es decir, rellenando los valores columna a columna!

Repaso NumPy (5)

¿Cómo es la matriz que imprimen de salida estas tres líneas de código?

```
a = np.arange(10).reshape(2,-1)
b = np.repeat(1, 10).reshape(2,-1)
np.concatenate([a, b], axis=0)
```

Lo concatenamos por filas!




```
In [36]: a
Out[36]:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

In [37]: b
Out[37]:
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

In [38]: np.concatenate([a, b], axis=0)
Out[38]:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

Repaso NumPy (6)

¿Qué resultado daría hacer A==B?

```
In [1]: A = [[1,2,3],[4,5,6],[7,8,9]]
   ...: B = [[1,2,-3],[10,5,1],[0,0,9]]

In [2]: A==B
```

Repaso NumPy (6)

¿Qué resultado daría hacer A==B?

```
In [1]: A = [[1,2,3],[4,5,6],[7,8,9]]
   ...: B = [[1,2,-3],[10,5,1],[0,0,9]]

In [2]: A==B
Out[2]: False
```

Repaso NumPy (6)

Pero… ¿y si lo declaramos como un NumPy array?

```
In [1]: import numpy as np
   ...: A = np.array([[1,2,3],[4,5,6],[7,8,9]])
   ...: B = np.array([[1,2,-3],[10,5,1],[0,0,9]])

In [2]: A==B
```

Repaso NumPy (6)

Pero… ¿y si lo declaramos como un NumPy array?

```
In [1]: import numpy as np
   ...: A = np.array([[1,2,3],[4,5,6],[7,8,9]])
   ...: B = np.array([[1,2,-3],[10,5,1],[0,0,9]])

In [2]: A==B
Out[2]:
array([[ True,  True, False],
       [False,  True, False],
       [False, False,  True]])
```

Repaso NumPy (7)

Tenemos un conjunto de datos *X*, con 200 filas (ejemplos) y 15 columnas (características/*features*). *y* incluye las etiquetas (salidas deseadas) para cada uno de los ejemplos. ¿De qué forma podríamos separar, de forma aleatoria (y eficiente, es decir, sin bucles *for*), minibatches disjuntos de 25 ejemplos?

```
import numpy as np

sampleSize = 200
featuresNumber = 15
minibatchSize = 25

X = np.random.uniform(-10,10, (sampleSize,featuresNumber))
y = np.random.randint(0,5,(sampleSize,1))
```

Repaso NumPy (7)

Tenemos un conjunto de datos *X*, con 200 filas (ejemplos) y 15 columnas (características/*features*). *y* incluye las etiquetas (salidas deseadas) para cada uno de los ejemplos. ¿De qué forma podríamos separar, de forma aleatoria (y eficiente, es decir, sin bucles *for*), minibatches disjuntos de 25 ejemplos?

```
import numpy as np

sampleSize = 200
featuresNumber = 15
minibatchSize = 25

X = np.random.uniform(-10,10, (sampleSize,featuresNumber))
y = np.random.randint(0,5,(sampleSize,1))

indexes = np.random.permutation(sampleSize)

minibatches_X = np.array_split(X[indexes], sampleSize // minibatchSize)
minibatches_y = np.array_split(y[indexes], sampleSize // minibatchSize)
```

https://numpy.org/doc/stable/reference/generated/numpy.array_split.html

Repaso NumPy (y 8)

Imaginemos que tenemos un conjunto de datos (x, con 5 filas/ejemplos y 3 columnas/*features*), unas etiquetas (y, salidas deseadas), y unos pesos (w, que representan los parámetros internos de nuestro modelo de aprendizaje automático). ¿Hay algún problema con el siguiente código para calcular el error cuadrático medio (MSE)?

```
def MSE(y,y_target):
    return (1/y.size)*np.linalg.norm(y-y_target)**2

x = np.array([[1, 2.5, 2.2],[1, 3.3, 1.2],[1, 6, 2],[1, 2, 9.1],[1, 5.5, 2.5]])
y = np.array([-1, 1, 1, -1, -1])
y.shape = (5,1)
w = np.array([1,2,3])
salidaObtenida = x.dot(w)

print('MSE: ',MSE(salidaObtenida,y))
```

Repaso NumPy (y 8)

```
def MSE(y,y_target):
    return (1/y.size)*np.linalg.norm(y-y_target)**2

x = np.array([[1, 2.5, 2.2],[1, 3.3, 1.2],[1, 6, 2],[1, 2, 9.1],[1, 5.5, 2.5]])
y = np.array([-1, 1, 1, -1, -1])
y.shape = (5,1)
w = np.array([1,2,3])
salidaObtenida = x.dot(w)

print('MSE: ',MSE(salidaObtenida,y))
```

```
y:  [[-1]
 [ 1]
 [ 1]
 [-1]
 [-1]]
```

El problema está en **`y.shape=(5,1)`**. Si visualizamos ese array, tiene la forma siguiente:

Esto hace que la diferencia entre la salida deseada y obtenida sea una matriz bidimensional, en lugar de un vector de errores.

```
salidaObtenida-y:  [[13.6 12.2 20.  33.3 20.5]
 [11.6 10.2 18.  31.3 18.5]
 [11.6 10.2 18.  31.3 18.5]
 [13.6 12.2 20.  33.3 20.5]
 [13.6 12.2 20.  33.3 20.5]]
```

Todo ello provoca que el MSE sea inesperadamente alto:

```
MSE:  2111.5799999999995
```

Repaso NumPy (y 8)

```
def MSE(y,y_target):
    return (1/y.size)*np.linalg.norm(y-y_target)**2

x = np.array([[1, 2.5, 2.2],[1, 3.3, 1.2],[1, 6, 2],[1, 2, 9.1],[1, 5.5, 2.5]])
y = np.array([-1, 1, 1, -1, -1])
y.shape = (5,1)
w = np.array([1,2,3])
salidaObtenida = x.dot(w)

print('MSE: ',MSE(salidaObtenida,y))
```

```
y.shape = (5,)
print('y: ', y)
print('salidaObtenida-y: ', salidaObtenida-y)
print('MSE: ',MSE(salidaObtenida,y))
```

Una forma de resolver el problema y calcular los errores adecuados, sería hacer **`y.shape=(5,)`**. De este modo, tendríamos lo siguiente:

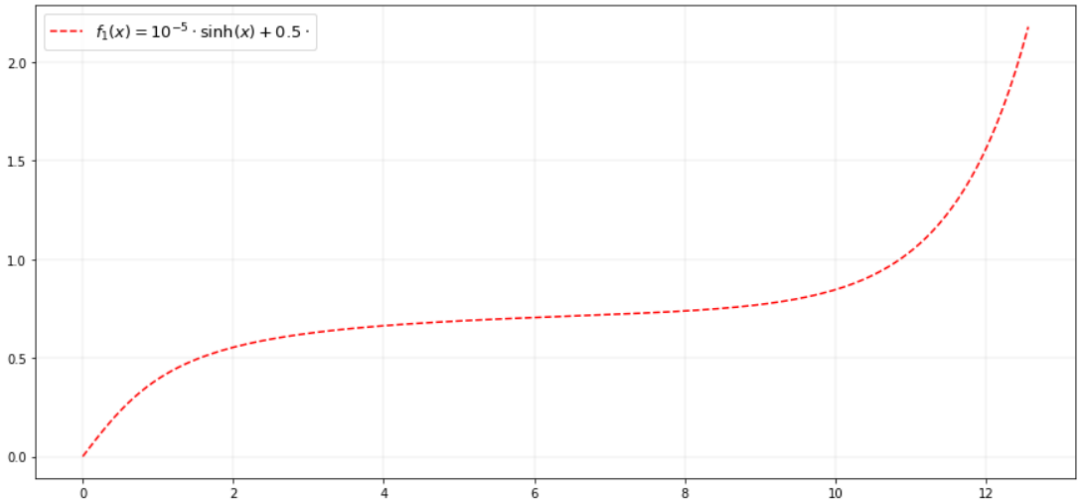
```
y:  [-1  1  1 -1 -1]
salidaObtenida-y:  [13.6 10.2 18.  33.3 20.5]
MSE:  428.4279999999999
```

Repaso Matplotlib (1)

¿Dónde está el problema de este código?

```
x = np.linspace(0, 4*np.pi, 150)
f1 = 10**(-5) * np.sinh(x) + 0.5 * np.arctan(x)

fig, ax = plt.subplots(figsize=(15,7))
ax.plot(x, f1, 'r--', label='$f_1(x)= 10^{-5}\cdot\sinh(x) + 0.5\cdot\arctan(x)$')
plt.grid(visible=True,linewidth=0.2)
plt.legend(fontsize=13)
plt.show()
```

```
<>:8: DeprecationWarning: invalid escape sequence \c
<>:8: DeprecationWarning: invalid escape sequence \c
<ipython-input-16-99396560ae76>:8: DeprecationWarning: invalid escape sequence \c
  ax.plot(x, f1, 'r--', label='$f_1(x)= 10^{-5}\cdot\sinh(x) + 0.5\cdot\arctan(x)$')
```



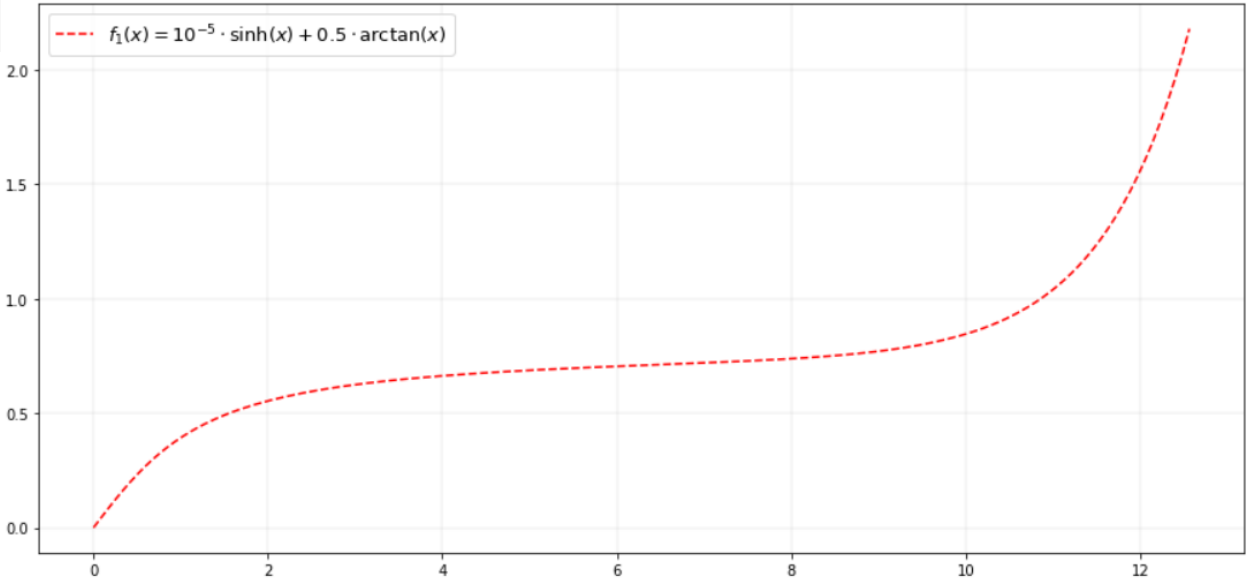
Repaso Matplotlib (1)

¿Dónde está el problema de este código?

```
x = np.linspace(0, 4*np.pi, 150)
f1 = 10**(-5) * np.sinh(x) + 0.5 * np.arctan(x)

fig, ax = plt.subplots(figsize=(15,7))
ax.plot(x, f1, 'r--', label='$f_1(x)= 10^{-5}\cdot\sinh(x) + 0.5\cdot\arctan(x)$')
plt.grid(visible=True,linewidth=0.2)
plt.legend(fontsize=13)
plt.show()
```

```
r'$f_1(x)= 10^{-5}\cdot\sinh(x) + 0.5\cdot\arctan(x)$'
```

Justo antes de la expresión en LaTeX (entre símbolos de $) debe ir una "r". Es lo que se llaman *raw strings* en Python. Sin la "r", el intérprete considera que "\a" es el carácter de escape y, por tanto, no te muestra la arcotangente en la expresión matemática. Sencillamente, llega a ese punto y dice: "Ah, me he encontrado un '\a', es decir, a partir de aquí no hay nada y, si lo hay, me lo salto." Con la "r" delante, ese símbolo se trata de modo literal y, por tanto, es procesado adecuadamente como texto LaTeX.



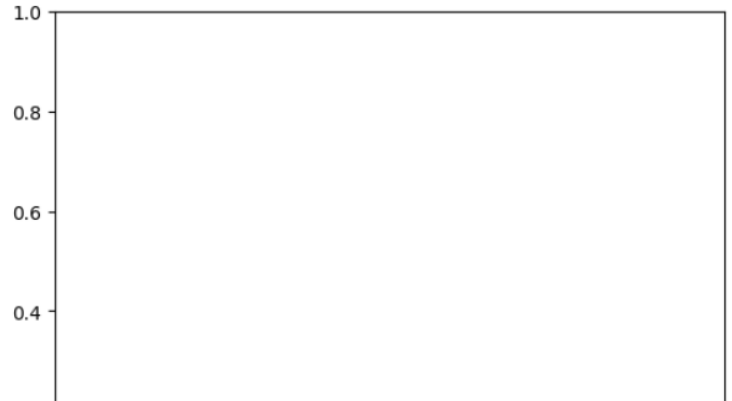
Repaso Matplotlib (y 2)

¿Dónde está el problema de este código?

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

x = np.linspace(-10, 10, 100)
y1 = np.exp(x)
y2 = np.log(x)

ax = plt.subplots()
ax.plot(x, y1, 'k--', label='y = exp(x)')
ax.plot(x, y2, 'k:', label='y = ln(x)')
legend = ax.legend(loc='lower right', shadow=True, fontsize='x-large')
ax.set_ylim((-5, 5))
legend.get_frame().set_facecolor('C0')
plt.grid()
plt.show()
```

```
<ipython-input-6-17c20ecb85ae>:6: RuntimeWarning: invalid value encountered in log
  y2 = np.log(x)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-17c20ecb85ae> in <cell line: 9>()
      7
      8 ax = plt.subplots()
----> 9 ax.plot(x, y1, 'k--', label='y = exp(x)')
     10 ax.plot(x, y2, 'k:', label='y = ln(x)')
     11 legend = ax.legend(loc='lower right', shadow=True, fontsize='x-large')

AttributeError: 'tuple' object has no attribute 'plot'
```

BUSCAR EN STACK OVERFLOW



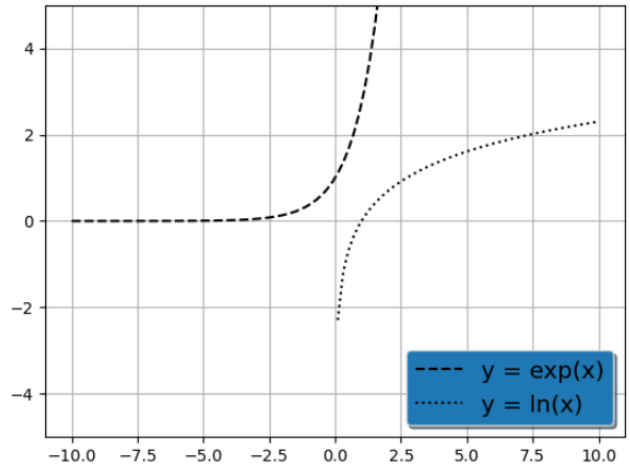
Repaso Matplotlib (y 2)

¿Dónde está el problema de este código?

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

x = np.linspace(-10, 10, 100)
y1 = np.exp(x)
y2 = np.log(x)

ax = plt.subplots()
ax.plot(x, y1, 'k--', label='y = exp(x)')
ax.plot(x, y2, 'k:', label='y = ln(x)')
legend = ax.legend(loc='lower right', shadow=True, fontsize='x-large')
ax.set_ylim((-5, 5))
legend.get_frame().set_facecolor('C0')
plt.grid()
plt.show()
```

Si no queremos obtener ese Warning, podríamos hacer
`import warnings`
`warnings.filterwarnings("ignore")`

```
<ipython-input-7-fba46653961b>:6: RuntimeWarning: invalid value encountered in log
  y2 = np.log(x)
```



`plt.subplots()` devuelve la figura y los ejes.

Returns:

fig : *Figure*

ax : *Axes* ***or array of Axes***

Deberíamos, por tanto, haber hecho:

```
_, ax = plt.subplots()
```

o

```
fig, ax = plt.subplots()
```

Prácticas de Aprendizaje Automático

Clase 2: Ejercicios/ejemplos de repaso

Pablo Mesejo y Salvador García
Universidad de Granada
Departamento de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE GRANADA