Números aleatorios

Numpy ofrece un conjunto de funciones y tipos para utilizar números aleatorios. Toda esta funcionalidad está agrupada en la subpaquete numpy.random.

El uso de números aleatorios y otras funciones asociadas es muy común en las Ciencias de Datos con varios objetivos entre los que podemos destacar: simular distribuciones para probar hipótesis, aumentar el tamaño de un dataset añadiendo nuevos datos sintéticos obtenidos como modificación de forma aleatoria a partir de los originales y también como una forma de muestrear conjuntos de datos para obtener particiones de los mismos.

Configuración del entorno.

Como siempre, comenzamos cargando el módulo y visualizando su versión ya que este dato es importante para consultar la documentación.

Además vamos a importar directamente el sub paquete numpy.random para facilitar la escritura de código. También aprovecharemos para indicar que a la hora de visualizar los datos se aproxime los números flotantes con 3 decimales.

En el momento de escribir este tutorial la versión de Numpy con la que se trabaja es 1.22.4.

In [1]:
```
import numpy as np
from numpy import random
print('Numpy version: {}'.format(np.__version__))
np.set_printoptions(precision=3)
```

```
Numpy version: 1.22.4
```

Consideraciones previas.

Un número aleatorio es una cantidad generada por un proceso que simula el azar. En una computadora sólo podemos aproximar el azar usando algoritmos, por lo que los números aleatorios generados en una computadora se denominan número *pseudo aleatorios*.

Existen distintos algoritmos para generar número aleatorios. El subpaque `numpy.random` ofrece varios, pero en este cuaderno nos centraremos en el generador de números por defecto de numpy que obtendremos con numpy.random.default_rng()

El generador utiliza un argumento `seed` (semilla) opcional que condiciona el primer número aleatorio generado. Si no se utiliza este parámetro, el generador se inicializa a su vez con un valor aleatorio lo cual hace que sea difícil predecir la secuencia de números aleatorios que puede generar.

En muchas ocasiones, por ejemplo, cuando depuramos un programa que use números aleatorios, necesitaremos poder reproducir casos de prueba. En esta situación necesitamos que la secuencia de

números aleatorios se repita de nuevo. Para ello podemos inicializar el generador con un valor de semilla (`seed`) fijo. De esta forma, la secuencia aleatoria generada se repetirá de nuevo.

En este cuaderno, con el objetivo de que los resultados de los ejercicios sean los mismos siempre, usaremos la semilla `seed=0`.

Generación simple de números aleatorios.

El generador ofrece varias funciones simples aleatorias. Se recomienda leer esta referencia antes de realizar los ejercicios.

Ejercicio 01: Simular 10 lanzamientos de un dado.

Cada lanzamiento se puede simular como un número aleatorio en el intervalo entero $[1, 6]$.

La salida debería ser algo parecido a esto:

```
Lanzamientos :  [6 4 4 2 2 1 1 1 2 5]
```

In [2]:
```
gen = random.default_rng(0) #Usamos semilla 0 para poder reproducir
lanzamientos = np.full(10, -1)
#Pon tu código aquí.
#Sugerencia: utiliza el método integers.

#
print('Lanzamientos : ',lanzamientos)
```

```
Lanzamientos :  [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Ejercicio 02: Simular 10 lanzamientos de una moneda "trucada", con una probabilidad de salir cara de un 60%. La salida será un array de valores lógicos donde `True` (sale cara), `False` (sale cruz).

Para simular un lanzamiento de esta moneda podemos generar un número aleatorio uniforme en el intervalo $n \in [0, 1)$ y decidir que el resultado es "cara" si $n > 0.4$.

La salida debería ser algo parecido a esto:

```
Lanzamientos :  [ True False False False  True  True  True  True  True
True]
```

In [3]:
```
gen = random.default_rng(0) #Usamos semilla 0 para poder reproducir
lanzamientos = np.full(10, False)
#Pon tu código aquí.
#Sugerencia: utiliza el método random. Usa el valor 0.6 para
#determinar si es cara o cruz.

#
print('Lanzamientos : ', lanzamientos)
```

```
Lanzamientos :  [False False False False False False False False False False]
```

Ejercicio 03: Simular el reparto, para cuatro jugadores, de 5 naipes de una baraja española con 40 naipes (10 naipes por palo, sin ochos ni nueves). Se asume que las cartas están numeradas de forma que el índice 0 es para As de Oros hasta 9 para el Rey de Oros, del 10 para el As de Copas hasta el 19 para el

Rey de Copas, y así sucesivamente para las Espadas y los Bastos, siendo el índice 39 el naipe Rey de Bastos.

Notar que en este caso no podemos simplemente generar número aleatorios en el rango entero $[0, 39]$ ya que se podrían repetir números, es decir, dos jugadores con el mismo naipe, lo cual no es posible. Este caso se simula con un muestro aleatorio sin reemplazo.

La salida debería ser algo parecido a esto:

```
Repartos:
 [[29  7  6 39 11]
 [33 21 34 14 20]
 [ 5 24 23  0 37]
 [ 2 17 16 26  1]]
```

In [4]:
```
gen = random.default_rng(0) #Usamos semilla 0 para poder reproducir
repartos = np.zeros((4, 5), 'd')
#Pon tu código aquí.
#Sugerencia: utiliza el método choice. Recuerda que los valores
#válidos de carta son el rango [0, 40) y que una vez repartida
#la carta a un jugador, esta ya no puede ser repartida a otro.

#
print('Repartos:\n', repartos)
```

```
Repartos:
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Permutaciones.

El generador ofrece dos funciones para realizar permutaciones aleatorias. Se recomienda leer esta referencia antes de realizar los ejercicios.

Estas funciones tienen muchas aplicaciones pero, centrándonos en las Ciencias de Datos y más concretamente en el "Aprendizaje Automático" a partir de los datos, muchos de los algoritmos empleados proporcionan resultados distintos según la orden de secuencia en que se le presentan los datos. Para obtener resultados más robustos, una técnica común es mostrar los datos repetidamente variando aleatoriamente el orden que se les presentan.

Ejercicio 04: Supongamos que tenemos una matriz donde las filas son muestras (individuos) de una distribución y las columnas son las características medidas para dichos individuos. Queremos obtener una distribución aleatoria distinta de las mismas muestras.

La salida debería ser algo parecido a esto:

```
Datos originales:
 [[0.637 0.27  0.041 0.017 0.813]
 [0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73 ]
 [0.176 0.863 0.541 0.3   0.423]]
Datos permutados:
 [[0.816 0.003 0.857 0.034 0.73 ]
 [0.176 0.863 0.541 0.3   0.423]
```

```
 [0.913 0.607 0.729 0.544 0.935]
 [0.637 0.27  0.041 0.017 0.813]]
```

In [5]:
```
gen = random.default_rng(0) #Usamos semilla 0 para poder reproducir
data = np.array(gen.random(4*5)).reshape(4,5)
data_p = np.zeros_like(data)
#Pon tu código aquí.
#Sugerencia: Prueba a obtener la solución comparando los
#métodos shuffle y permutation.

#
print('Datos originales:\n', data)
print('Datos permutados:\n', data_p)
```

```
Datos originales:
 [[0.637 0.27  0.041 0.017 0.813]
 [0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73 ]
 [0.176 0.863 0.541 0.3   0.423]]
Datos permutados:
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Generar números aleatorios siguiendo una distribución determinada.

Los números pseudo aleatorios obtenidos hasta el momento siguen una distribución uniforme, es decir, cualquier valor en el intervalo $[0, 1)$ para `random()` o el intervalo entero indicado para `intergers()` tiene igual probabilidad de ser generado.

El generador ofrece la posibilidad de obtener números aleatorios siguiendo otras distribuciones de probabilidad distinta a la uniforme. Se recomienda leer esta referencia antes de realizar los ejercicios.

Ejercicio 05: Supongamos que tenemos un dataset representado por una matriz donde las filas son muestras (individuos) de una población y, las columnas son las características medidas para dichos individuos.

Queremos aumentar nuestro dataset obteniendo una nueva versión de los datos a los que se les añade ruido gaussiano con media 0 y desviación ρ_j donde j indica la columna (característica) correspondiente. El valor ρ_j para cada característica se calcula como una fracción n_j de la desviación de dicha característica σ_j en los datos originales. Esto es:

$$data\_a[i, j] = data[i, j] + r, r \sim N(0, \rho_j)$$

$$\rho_j = n_j \sigma_j$$

La salida debería ser algo parecido a esto:

```
Datos originales:
 [[0.637 0.27  0.041 0.017 0.813]
 [0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73 ]
 [0.176 0.863 0.541 0.3   0.423]]
Ruido a añadir:
 [[-0.004  0.045 -0.021  0.008  0.017]
```

```
 [ 0.003 -0.024 -0.029 -0.01   0.004]
 [-0.029 -0.007 -0.005  0.012  0.004]
 [ 0.01  -0.021 -0.004  0.017  0.028]]
Datos aumentados:
 [[ 0.633  0.314  0.02   0.024  0.83 ]
 [ 0.915  0.582  0.701  0.534  0.939]
 [ 0.787 -0.004  0.852  0.045  0.734]
 [ 0.186  0.842  0.537  0.317  0.451]]
```

In [6]:

```
gen = random.default_rng(0) #Usamos semilla 0 para poder reproducir
data = np.array(gen.random(4*5)).reshape(4,5)
ruido = np.zeros_like(data)

#Pon tu código aquí.
#Sugerencia: Utiliza el método normal() para obtener números
#siguiendo una distribución gaussiana.
#Para simplificar, todas las desviaciones para generar el ruido
# en cada característica (columna) serán la fracción 1/10 de la
#correspondiente desviación en los datos originales.
#Recuerda no usar bucles.

#

data_a = data + ruido
print('Datos originales:\n', data)
print('Ruido a añadir:\n', ruido)
print('Datos aumentados:\n', data_a)
```

```
Datos originales:
 [[0.637 0.27  0.041 0.017 0.813]
 [0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73 ]
 [0.176 0.863 0.541 0.3   0.423]]
Ruido a añadir:
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
Datos aumentados:
 [[0.637 0.27  0.041 0.017 0.813]
 [0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73 ]
 [0.176 0.863 0.541 0.3   0.423]]
```

In [ ]: