Métodos de agregación

Una operación muy común a realizar sobre un conjunto de datos es calcular valores que resuman el conjunto como pueden ser medias, varianzas, valores mínimo o máximos, etc.

Numpy ofrece un conjunto de operaciones con este propósito y son el objeto de estudio de este cuaderno. Este cuaderno se basa en la documentación oficial que puedes consultar aquí.

Configuración del entorno.

Como siempre, comenzamos cargando el módulo y visualizando su versión ya que este dato es importante para consultar la documentación. 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
print('Numpy version: {}'.format(np.__version__))
np.set_printoptions(precision=3)
from io import StringIO
```

```
Numpy version: 1.22.4
```

Consideraciones generales.

Como ya hemos comentado Numpy cuenta con una amplia variedad de funciones para agregar datos. Como es posible que alguno de los datos a ser agregado sea un valor `np.nan` ("*not a number*"), en general, el resultado tras agregarlo sería un valor `np.nan`. Para protegernos de posibles valores `np.nan`, si estimamos que dichos valores pueden ocurrir, para cada función de agregación que estudiemos existe una versión "segura" que filtra los valores `np.nan` y no los agrega. Si tenemos una operación `np.oper()`, la versión segura será `np.nanoper()`. El precio a pagar es que esta versión segura de la operación es menos eficiente.

Otra consideración general a tener en cuenta es que todas las funciones de agregación tienen un argumento opcional `axis` con valor por defecto `None`. Este valor por defecto implica que al aplicar la operación sobre el ndarray se realiza sobre una vista `aplanada` ("*flattened*") del mismo devolviendo un valor escalar.

De manera opcional, podemos usar este argumento para indicar sobre cuál eje (dimensión) se debe realizar la agregación, por ejemplo, si tenemos una matriz, podríamos agregar por filas con `axis=0` o por columnas con `axis=1`.

Por último, la mayoría de las funciones que vamos a estudiar pueden ser utilizadas como una función o como un método de un objeto ndarray con el mismo significado, es decir, si `a` es ndarray, podemos utilizar

tanto `np.operacion(a)` como `a.operacion()` con el mismo significado.

Agregando para obtener la sumas y productos.

Podemos agregar los datos de un ndarray sumando todos sus elemenos con sum() o multiplicando todos sus elementos con prod() o sus respectivas versiones seguras nansum() y nanprod().

Ejercicio 1: Supongamos que una matriz `CM` representa la tabla de confusión generada por un algoritmo de clasificación. Se desea implementar una función que normalice dicha matriz, es decir, que la suma de todos sus elementos sea 1.0. Esto se puede hacer aplicando la expresión:

$$\mathrm{CM}' = \frac{1}{\alpha}\,\mathrm{CM} \text{ con } \alpha = \sum_{r,c} \mathrm{CM}[i,j]$$

La salida debería ser parecida a lo siguiente:

```
Matriz original:
 [[ 5  3  0]
 [ 2  3  1]
 [ 0  2 11]]
Matriz normalizada:
 [[0.185 0.111 0.   ]
 [0.074 0.111 0.037]
 [0.    0.074 0.407]]
```

In [2]:
```
def normalizar_matriz(m):
    '''
    Devuelve una versión normalizada de la entrada, de tal forma
    que la suma de todos sus elementos sea igual a 1. Se asume
    una matriz con elementos positivos no nula.
    '''
    ret = np.zeros_like(m)
    #Pon tu código aquí.
    #Sugerencia: usa la función np.sum() para agregar la suma de
    #todos los elementos.
    #Recuerda impedir dividir por cero.

    #
    return ret

CM = np.array([[5, 3, 0],
               [2, 3, 1],
               [0, 2, 11]])
print('Matriz original:\n', CM)
print('Matriz normalizada:\n', normalizar_matriz(CM))
```

```
Matriz original:
 [[ 5  3  0]
 [ 2  3  1]
 [ 0  2 11]]
Matriz normalizada:
 [[0 0 0]
 [0 0 0]
 [0 0 0]]
```

Ejercicio 2: Supongamos que una matriz `CM` representa la tabla de confusión generada por un algoritmo de clasificación. Se desea implementar una función que normalice dicha matriz de forma que las filas sumen uno.

La salida debería ser parecida a lo siguiente:

```
Matriz original:
 [[ 5  3  0]
 [ 2  3  1]
 [ 0  2 11]]
Matriz normalizada por filas:
 [[0.625 0.375 0.   ]
 [0.333 0.5   0.167]
 [0.    0.154 0.846]]
```

In [3]:
```
def normalizar_matriz_filas(m):
    '''
    Devuelve una versión normalizada de la entrada, de tal forma
    que la suma de todos sus elementos por filas sea igual a 1.
    '''
    ret = np.zeros_like(m)
    #Pon tu código aquí.
    #Sugerencia: usa la función np.sum() indicando con el argumento
    #axis que queremos agregar por filas.
    #Recuerda que puedes reconfigurar la forma con ndarray.reshape()
    #Evita utilizar bucles.

    #
    return ret

CM = np.array([[5, 3, 0],
               [2, 3, 1],
               [0, 2, 11]])
print('Matriz original:\n', CM)
print('Matriz normalizada por filas:\n',
      normalizar_matriz_filas(CM))
```

```
Matriz original:
 [[ 5  3  0]
 [ 2  3  1]
 [ 0  2 11]]
Matriz normalizada por filas:
 [[0 0 0]
 [0 0 0]
 [0 0 0]]
```

Ejercicio 3: Supongamos que una matriz `CM` representa la tabla de confusión generada por un algoritmo de clasificación. Se desea implementar una función que normalice dicha matriz de forma que las columnas sumen uno.

La salida debería ser parecida a lo siguiente:

```
Matriz original:
 [[ 5  3  0]
 [ 2  3  1]
 [ 0  2 11]]
Matriz normalizada por columnas:
 [[0.714 0.375 0.   ]
 [0.286 0.375 0.083]
 [0.    0.25  0.917]]
```

In [4]:
```
def normalizar_matriz_columnas(m):
    '''
    Devuelve una versión normalizada de la entrada, de tal forma
    que la suma de todos sus elementos por columnas sea igual a 1.
```

```
    '''
    ret = np.zeros_like(m)
    #Pon tu código aquí.
    #Sugerencia: usa la función np.sum() indicando con el argumento
    #axis que queremos agregar por filas.
    #Recuerda que puedes reconfigurar la forma con ndarray.reshape()
    #Evita utilizar bucles.

    #
    return ret

CM = np.array([[5, 3, 0],
               [2, 3, 1],
               [0, 2, 11]])
print('Matriz original:\n', CM)
print('Matriz normalizada por columnas:\n',
      normalizar_matriz_columnas(CM))
```

```
Matriz original:
 [[ 5  3  0]
 [ 2  3  1]
 [ 0  2 11]]
Matriz normalizada por columnas:
 [[0 0 0]
 [0 0 0]
 [0 0 0]]
```

Agregando para obtener promedios y varianzas.

Numpy ofrece un conjunto de operaciones para obtener datos estadísticos agregados como medias, varianzas, percentiles, etc. Se recomienda leer la documentación sobre estas funciones en esta referencia.

Ejercicio 4: Supongamos que matriz `img` representa una imagen monocroma cuyos elementos son los valores de luz adquirida en cada elemento del sensor ("*pixel*") en el rango entero $[0, 255]$. Queremos definir una función que nos indique si la imagen es oscura (el valor $medio \leq 85$), normal (el valor $85 < medio \leq 170$) o clara (el valor $medio > 170$), devolviendo los valores 0, 1 o 2 respectivamente.

La salida debería ser parecida a lo siguiente:

```
Imagen A:
 [[163 110 132 158]
 [116  87 105 138]
 [125 130 144 115]
 [157 147 152 151]]
Tipo: normal
Imagen B:
 [[99 46 68 94]
 [52 23 41 74]
 [61 66 80 51]
 [93 83 88 87]]
Tipo: oscura
Imagen C:
 [[227 174 196 222]
 [180 151 169 202]
 [189 194 208 179]
 [221 211 216 215]]
Tipo: clara
```

In [5]:
```
def tipo_de_imagen(img):
```

```
    '''
    Dada la matriz img, que representa una imagen monocroma cuyos
    elementos son los valores de luz adquirida en cada elemento
    del sensor ("*pixel*") en el rango entero $[0, 255]$.
    Devolver un entero que indica si la imagen es oscura
    (el valor $medio <= 85$), normal (el valor $85<medio<170$) o
    clara (el valor $medio >170$),
    devolviendo los valores 0, 1 o 2 respectivamente.
    '''
    tipo = 0
    #Pon tu código aquí.
    #Sugerencia: usa la función np.mean() para calcular el valor
    #medio.

    #
    return tipo

tipos = ['oscura', 'normal', 'clara']
img_a = np.array([[163, 110, 132, 158],
                  [116,  87, 105, 138],
                  [125, 130, 144, 115],
                  [157, 147, 152, 151]])
print('Imagen A:\n', img_a)
print('Tipo: {}'.format(tipos[tipo_de_imagen(img_a)]))
img_b = img_a - 64
print('Imagen B:\n', img_b)
print('Tipo: {}'.format(tipos[tipo_de_imagen(img_b)]))
img_c = img_a + 64
print('Imagen C:\n', img_c)
print('Tipo: {}'.format(tipos[tipo_de_imagen(img_c)]))
```

```
Imagen A:
 [[163 110 132 158]
 [116  87 105 138]
 [125 130 144 115]
 [157 147 152 151]]
Tipo: oscura
Imagen B:
 [[99 46 68 94]
 [52 23 41 74]
 [61 66 80 51]
 [93 83 88 87]]
Tipo: oscura
Imagen C:
 [[227 174 196 222]
 [180 151 169 202]
 [189 194 208 179]
 [221 211 216 215]]
Tipo: oscura
```

Ejercicio 5: Supongamos que matriz `img` representa una imagen monocroma cuyos elementos son los valores de luz adquirida en cada elemento del sensor ("*pixel*") en el rango entero $[0, 255]$. Queremos definir una función que nos indique el contraste de la imagen. Para ello calcularemos el coeficiente de variación que se define como $cv = 100 * \sigma/\mu$ donde μ es le valor medio de luz y σ su desviación. Devolver un entero que indica si la imagen está poco contrastada si el $cv < 5$, con un contraste normal si $5 \leq cv < 10$, o muy contrastada si $cv \geq 10$ devolviendo los valores 0, 1, 2 respectivamente.

La salida debería ser parecida a lo siguiente:

```
Imagen A:
 [[163 110 132 158]
 [116  87 105 138]
 [125 130 144 115]
```

```
 [157 147 152 151]]
Tipo: muy contrastada
Imagen B:
 [[209 183 194 207]
 [186 171 180 197]
 [190 193 200 185]
 [206 201 204 203]]
Tipo: contraste normal
Imagen C:
 [[168 155 161 167]
 [157 149 154 162]
 [159 160 164 156]
 [167 164 166 165]]
Tipo: poco contraste
```

In [6]:

```python
def contraste_de_imagen(img):
    '''
    Dada la matriz img, que representa una imagen monocroma cuyos
    elementos son los valores de luz adquirida en cada elemento
    del sensor ("*pixel*") en el rango entero $[0, 255]$.
    Devolver un entero que indica si la imagen está poco
    contrastada si el coeficiente de variación (en porcentaje) cv<5,
    con un contraste normal si 5<=cv<10, o muy contrastada si
    cv>=10% devolviendo los valores 0, 1, 2 respectivamente.
    '''
    tipo = 0
    #Pon tu código aquí.
    #Sugerencia: usa la función np.mean()y np.std() para calcular
    #el valor medio y la desviación.

    #
    return tipo

tipos = ['poco contraste', 'contraste normal', 'muy contrastada']
img_a = np.array([[163, 110, 132, 158],
                  [116,  87, 105, 138],
                  [125, 130, 144, 115],
                  [157, 147, 152, 151]])
print('Imagen A:\n', img_a)
print('Tipo: {}'.format(tipos[contraste_de_imagen(img_a)]))
img_b = 128+img_a//2
print('Imagen B:\n', img_b)
print('Tipo: {}'.format(tipos[contraste_de_imagen(img_b)]))
img_c = 128+img_a//4
print('Imagen C:\n', img_c)
print('Tipo: {}'.format(tipos[contraste_de_imagen(img_c)]))
```

```
Imagen A:
 [[163 110 132 158]
 [116  87 105 138]
 [125 130 144 115]
 [157 147 152 151]]
Tipo: poco contraste
Imagen B:
 [[209 183 194 207]
 [186 171 180 197]
 [190 193 200 185]
 [206 201 204 203]]
Tipo: poco contraste
Imagen C:
 [[168 155 161 167]
 [157 149 154 162]
 [159 160 164 156]
```

```
 [167 164 166 165]]
Tipo: poco contraste
```

Ejercicio 6: Supongamos que la matriz `data` representa un dataset donde cada fila representa un vector de características de un individuo de la población origen del dataset. Una operación muy utilizada es normalizar el dataset de forma que cada característica (columna de la matriz) esté normalizada con media 0 y varianza 1.

Para ello se debe aplicar a cada característica j, de la muestra i la operación:

$$\text{data}[i,j] = \frac{\text{data}[i,j] - \mu_j}{\sigma_j}$$

donde μ_j y σ_j son los valores media y desviación para la columna j.

Se desea implementar una función que realice esta normalización.

La salida debería ser parecida a lo siguiente:

```
Datos sin normalizar:
 [[0 7 9 5]
 [2 1 7 2]
 [0 6 3 5]]
Datos normalizados:
 [[-0.707  0.889  1.069  0.707]
 [ 1.414 -1.397  0.267 -1.414]
 [-0.707  0.508 -1.336  0.707]]
```

In [7]:
```python
def normalizar_dataset_media_varianza(data):
    '''
    Normalizar cada columna de a para que tenga
    media=0.0, varianza=1.0.
    '''
    data_n = np.zeros_like(data)
    #Pon tu código aquí.
    #Sugerencia: usa la función np.mean()y np.std() para calcular
    #el valor medio y la desviación por columnas. Usa el valor
    #apropiado para el argumento axis para agrupar.

    #
    return data_n

data = np.array([[0, 7, 9, 5],
                 [2, 1, 7, 2],
                 [0, 6, 3, 5]])
print('Datos sin normalizar:\n', data)
print('Datos normalizados:\n',
      normalizar_dataset_media_varianza(data))
```

```
Datos sin normalizar:
 [[0 7 9 5]
 [2 1 7 2]
 [0 6 3 5]]
Datos normalizados:
 [[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Agregando para obtener estadísticos de orden.

Numpy proporciona varias funciones para obtener valores mínimos, máximos, percentiles, etc. Se recomienda leer estas dos referencias 1 y 2 antes de realizar los ejercicios.

Ejercicio 7: Supongamos que la matriz `data` representa un dataset donde cada fila representa un vector de características de un individuo de la población origen del dataset. Queremos normalizar el dataset de forma que todos las características estén evaluadas en el intervalo $[a, b]$ donde el valor a se asignará al valor mínimo de esa característica y b al máximo. Es decir, realizar la operación:

$$\text{data}[i,j] = a + \frac{\text{data}[i,j] - min_j}{max_j - min_j} \times (b - a)$$

donde min_j y max_j son los valores mínimo y máximo para la columna j.

Se desea definir una función para realizar esa normalización.

La salida debería ser parecida a lo siguiente:

```
Datos sin normalizar:
 [[0 7 9 5]
 [2 1 7 2]
 [0 6 3 5]]
Datos normalizados en el intervalo [-1,1]:
 [[-1.     1.     1.     1.   ]
 [ 1.    -1.     0.333 -1.   ]
 [-1.     0.667 -1.     1.   ]]
```

In [8]:
```
def normalizar_dataset_min_max(data, a=0.0, b=1.0):
    '''
    Normalizar cada columna al intervalo [a,b].
    '''
    data_n = np.zeros_like(data)
    #Pon tu código aquí.
    #Sugerencia: usa la función np.amin()y np.amax(). Usa el valor
    #apropiado para el argumento axis para agrupar.

    #
    return data_n

data = np.array([[0, 7, 9, 5],
                 [2, 1, 7, 2],
                 [0, 6, 3, 5]])
print('Datos sin normalizar:\n', data)
a=-1
b=1
print('Datos normalizados en el intervalo [{},{}]:\n'.format(a,b),
      normalizar_dataset_min_max(data, a ,b))
```

```
Datos sin normalizar:
 [[0 7 9 5]
 [2 1 7 2]
 [0 6 3 5]]
Datos normalizados en el intervalo [-1,1]:
 [[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Agregando valores lógicos.

Hay dos operaciones que son útiles para agregar valores lógicos. La primera np.any() permite comprobar si

algún valor es `True` en un ndarray. La segunda operación np.all() comprueba si todos los valores de un ndarray son `True`.

Ejercicio 8: Tenemos un dataset desde un fichero CSV y es conocido la existencia de valores perdidos. Ya sabemos que estos valores perdidos se indicarán con el valor `np.nan`. Se quiere cargar dicho dataset y detectar que muestras (filas) tienen algún valor `np.nan` mediante un array de valores lógicos.

La salida debería ser parecida a lo siguiente:

```
Dataset:
 [[ 1.  2.  3.  4.]
 [ 5. nan  7.  8.]
 [ 9. 10. nan 12.]]
Muestras inválidas:  [False  True  True]
```

In [10]:
```
def detectar_muestras_invalidas(data):
    '''
    Devuelve un array con un valor lógico por fila indicando si
    en dicha fila hay algún valor np.nan.
    '''
    ret_v = np.full(data.shape[0], False)
    #Pon tu código aquí.
    #Sugerencia: usa la ufunc np.isnan para detectar valores nan
    # y la función np.any para agregar. Usa el argumento axis
    # para obtener un valor agrupado para cada fila.

    #
    return ret_v


file = StringIO(
u'''
1,2,3,4
5,,7,8
9,10,,12
''')
data = np.array([[0]])
#Pon tu código aquí.
#Cargar el dataset de forma que los valores perdidos se
#marquen con np.nan.

#
print('Dataset: \n',data)
print('Muestras inválidas: ', detectar_muestras_invalidas(data))
```

```
Dataset:
 [[0]]
Muestras inválidas:  [False]
```

Ejercicio 9: Tenemos un dataset en el cual las filas son muestras y las columnas las características medidas de cada individuo. Se desea filtrar muestras cuyos individuos tengan alguna característica con un valor raro ("*outlier*"). Se considera que una muestra i es "rara" cuando para alguna de sus características j se cumple:

$$|data[i,j] - \mu_j| > k\sigma_j$$

siendo μ_j y σ_j las media y desviación para la columna j y k un valor dado.

La salida debería ser parecida a lo siguiente:

```
Dataset:
 [[ 1.  2.  3.]
 [ 1.  2.  3.]
 [-1.  2.  3.]
 [ 1.  2.  3.]
 [ 1.  2.  3.]
 [ 1.  1.  3.]
 [ 1.  2.  3.]
 [ 1.  2.  0.]
 [ 1.  2.  3.]
 [ 1.  2.  3.]]
Muestras raras:  [False False  True False False  True False  True False
False]
```

In [11]:

```python
def detectar_muestras_raras(data, k=2.5):
    '''
    Devuelve un array con un valor lógico por fila indicando si
    en dicha fila corresponde a una muestra "rara".
    '''
    ret_v = np.full(data.shape[0], False)
    #Pon tu código aquí.
    #Sugerencia: usa las funciones de agregación para obtener las
    #medias y las varianzas de las columnas.
    #Usa las funciones universales necesarias de forma que
    #no se necesiten bucles.

    #
    return ret_v


data = np.array([[ 1.,  2.,  3.],
                 [ 1.,  2.,  3.],
                 [-1.,  2.,  3.],
                 [ 1.,  2.,  3.],
                 [ 1.,  2.,  3.],
                 [ 1.,  1.,  3.],
                 [ 1.,  2.,  3.],
                 [ 1.,  2.,  0.],
                 [ 1.,  2.,  3.],
                 [ 1.,  2.,  3.]])
print('Dataset: \n',data)
print('Muestras raras: ', detectar_muestras_raras(data))
```

```
Dataset:
 [[ 1.  2.  3.]
 [ 1.  2.  3.]
 [-1.  2.  3.]
 [ 1.  2.  3.]
 [ 1.  2.  3.]
 [ 1.  1.  3.]
 [ 1.  2.  3.]
 [ 1.  2.  0.]
 [ 1.  2.  3.]
 [ 1.  2.  3.]]
Muestras raras:  [False False False False False False False False False False]
```

In [ ]: