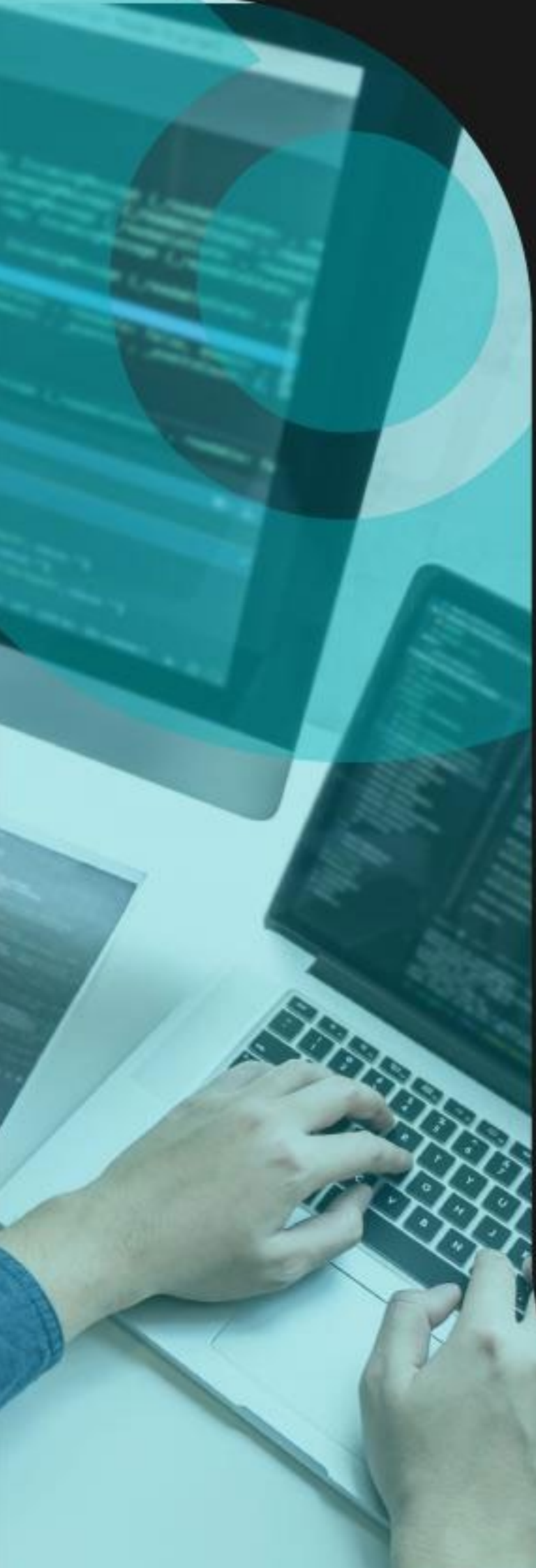


AWAKELAB

BASECAMP

Ciencia de Datos

Módulo: Análisis Exploratorio y Programación Estadística

Aprendizaje Esperado

5. Presenta información agregada utilizando funciones de agrupación, agregación y pivoteo para el análisis de un set de datos

Agrupación de datos

Definición Técnica: *Los datos agrupados son aquellos que están clasificados en función a un criterio, mostrando una frecuencia para cada clase o grupo formado.*

Es decir, los **datos** agrupados están separados por categorías, y cada dato u observación solo puede pertenecer a una categoría (no a dos o más).

Debemos recordar que un dato estadístico es la representación de una variable cualitativa o cuantitativa. Esto, mediante la asignación de un número, letra o símbolo. Otro asunto importante a mencionar es que, de acuerdo con las fuentes revisadas, se suelen usar datos agrupados cuando se trata de muestras de más de 20 datos. En cambio, con muestras más pequeñas, no es tan necesario agrupar, de todos modos siempre dependerá de los datos que se deseen estudiar o graficar.

El objetivo fundamental de agrupar los datos es que el análisis de los mismos pueda ser más sencillo, de manera que se pueda hacer una primera aproximación a los resultados de forma rápida.

Además, la agrupación sirve para poder elaborar, a partir de la información recopilada, herramientas visuales como un **histograma**, un **gráfico de barras** o un **gráfico circular**.

Diferencia entre datos agrupados y no agrupados

La principal **diferencia entre datos agrupados y no agrupados** es que los primeros han sido divididos por categorías, como habíamos mencionado. En cambio, los datos no agrupados se presentan tal cual han sido recopilados, sin ninguna modificación (quizás pueden ser ordenados, por ejemplo, de menor a mayor).

Como también habíamos señalado previamente, el tamaño de la muestra es un factor a tomar en cuenta. Si tuviéramos mil datos, por ejemplo, lo más práctico sería resumir la información en una tabla donde se observa la **distribución de frecuencias**.

Debemos recordar que la distribución de frecuencias es la forma en la que un conjunto de datos se clasifica en distintos grupos excluyentes entre sí. Es decir, si un dato pertenece a un grupo no puede ser parte de otro.

Ejemplo de datos agrupados

Un ejemplo de datos agrupados sería el siguiente, donde hemos resumido la información sobre los ingresos mensuales de un grupo de personas.

Ingresos mensuales	Frecuencia
[1.500-2.500]	120
(2.500-3.500]	210
(3.500-4.500]	300
(4.500-5.500]	250
(5.500-6.500]	400
(6.500-7.500]	510
(7.500-8.500]	420
(8.500-9.500]	416
(9.500-10.500]	100

En la tabla podemos observar que, por ejemplo, 210 personas de la muestra tienen ingresos mensuales de entre 2.500 y 3.500 euros.

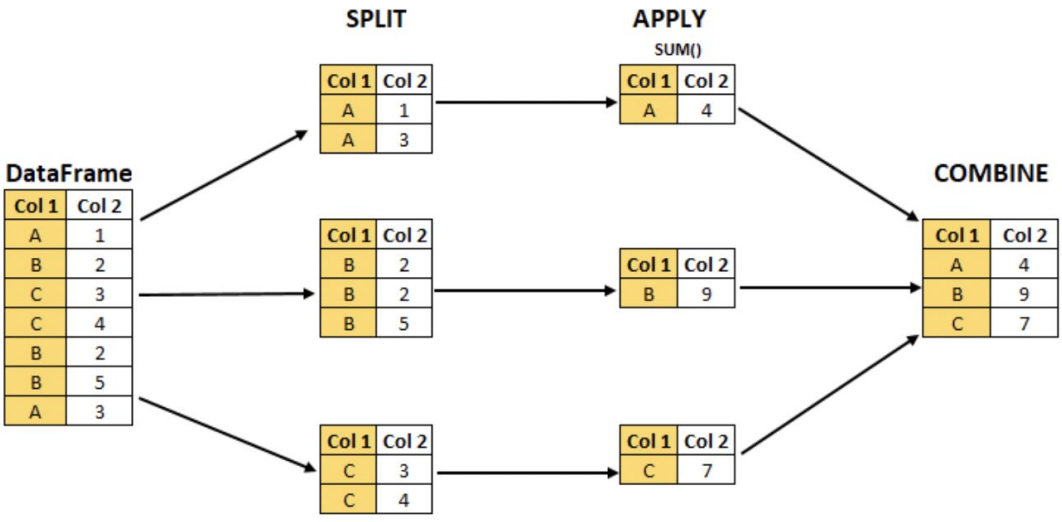
Ahora, si los datos no estuvieran agrupados simplemente se mostrarían como en un listado.

Nombre	Ingresos
Samuel Sánchez	2.700
Alex Benavides	3.100
Ernesto Jiménez	4.500
Alicia Fernández	6.100
Beatriz Borja	2.900
Vilma Zapata	3.400

Estrategia de división, aplicación y combinación - split-apply-combine

En esta parte usaremos funciones que nos permitan organizar nuestra información para hacer posteriores análisis estadísticos. Para ello, existe la estrategia split-apply-combine en la que dividimos un gran problema en pequeñas partes manejables (Split), operamos en cada parte de forma independiente (Apply) y luego juntamos todas las partes (Combine). Muchas herramientas existentes pueden usar split-apply-combine mediante la función GroupBy en SQL y Python, LOD en Tableau y mediante el uso de funciones plyr en R, por nombrar algunas. En este artículo, no analizaremos solo la implementación de esta estrategia, sino que también veremos algunas aplicaciones relevantes de esta estrategia en la ingeniería de funciones.

En Python hacemos esto usando GroupBy e involucra uno o más de los tres pasos de la estrategia split-apply-combine. Comencemos definiendo cada uno de los tres pasos:



Uso de la función Split-Apply-Combine

1. **Split**: dividir los datos en grupos según algunos criterios, creando así un objeto GroupBy. (Podemos usar la columna o una combinación de columnas para dividir los datos en grupos).

2. **Apply**: Aplicar una función a cada grupo de forma independiente. (Agrega, transforma o filtra los datos en este paso).
3. **Combine**: Combina los resultados en una estructura de datos (Pandas Series, Pandas DataFrame).

Conjunto de datos

Para profundizar un poco más, vamos a crear un dato ficticio para que sirva como ejemplo. Examine detenidamente el marco de datos (data_sales), que se proporciona a continuación.

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

sales_dict={'colour' : ['Yellow', 'Black',
'Blue','Red', 'Yellow', 'Black', 'Blue', 'Red',
'Yellow', 'Black', 'Blue', 'Red', 'Yellow', 'Black',
'Blue', 'Red', 'Blue', 'Red'],

           'sales' : [100000, 150000, 80000, 90000,
200000, 145000, 120000, 300000, 250000, 200000,
160000, 90000, 90100, 150000, 142000, 130000, 400000,
350000],

           'transactions' : [100, 150, 820, 920, 230,
120, 70, 250, 250, 110, 130, 860, 980, 300, 150, 170,
230, 280],

           'product' : ['type A', 'type A', 'type A',
'type A', 'type A', 'type A', 'type A', 'type A',
'type A', 'type B', 'type B', 'type B', 'type B',
'type B', 'type B', 'type B', 'type B', 'type B']}
```

```
data_sales=pd.DataFrame(sales_dict)

data_sales
```

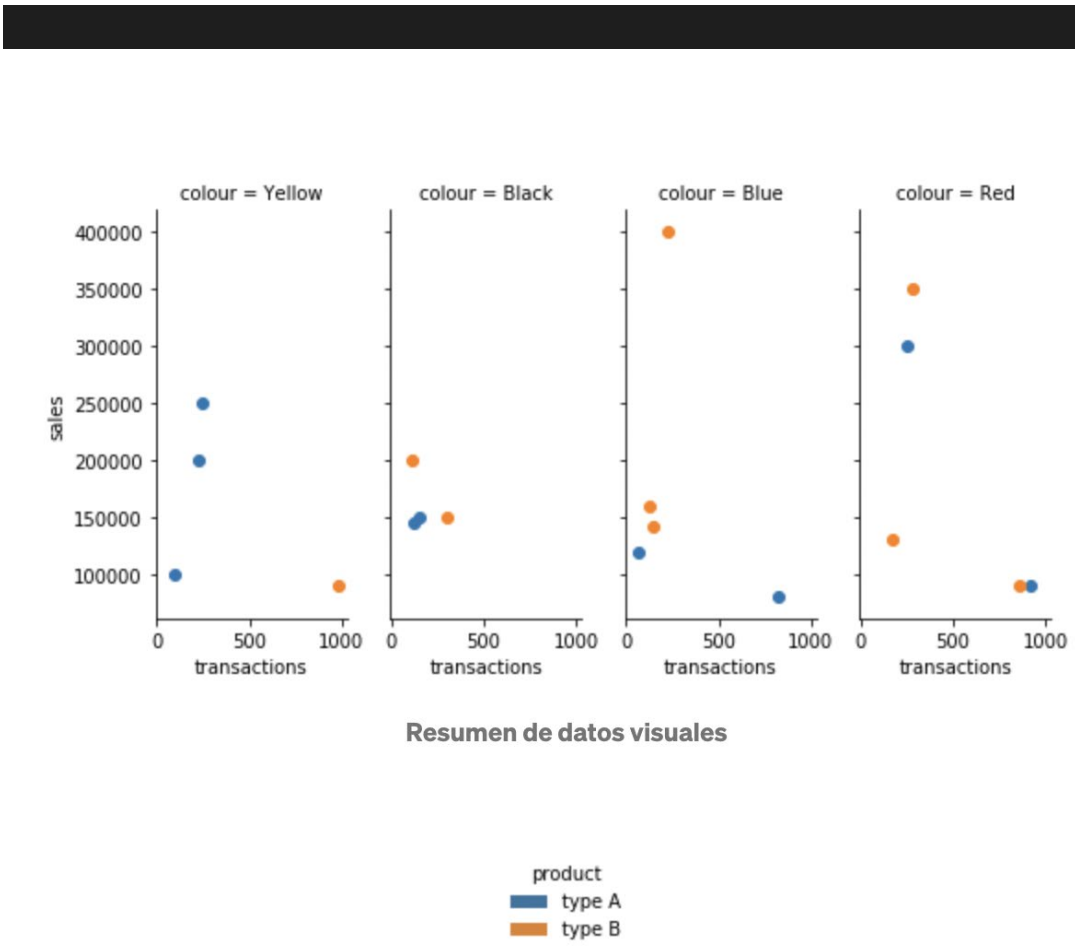
Obteniendo la siguiente tabla de datos:

	colour	sales	transactions	product
0	Yellow	100000	100	type A
1	Black	150000	150	type A
2	Blue	80000	820	type A
3	Red	90000	920	type A
4	Yellow	200000	230	type A
5	Black	145000	120	type A
6	Blue	120000	70	type A
7	Red	300000	250	type A
8	Yellow	250000	250	type A
9	Black	200000	110	type B
10	Blue	160000	130	type B
11	Red	90000	860	type B
12	Yellow	90100	980	type B
13	Black	150000	300	type B
14	Blue	142000	150	type B
15	Red	130000	170	type B
16	Blue	400000	230	type B
17	Red	350000	280	type B

Para resumir todos los datos, se ha utilizado la biblioteca Seaborn para crear una visualización, que incluye todos los datos gráficamente.

```
graph = sns.FacetGrid(data_sales, col="colour",
height=4, hue="product",aspect=.5)
```

```
graph.map(plt.scatter, "transactions", "sales");
```



Resumen de datos visuales

Después de crear y resumir los datos, como primer paso pasamos a la primera parte de Split-Apply-Combine.

SPLIT: crea un objeto.

En este paso, crearemos los grupos a partir del marco de datos 'data_sales' agrupando sobre la base de la columna 'color'.

```
# Split: agrupa la columna "colour"


data_gby = data_sales.groupby('colour')
```

```
print(type(data_gby))
```

Una vez que aplicamos la función groupby() en el marco de datos, crea el **Groupby object** como resultado.Podemos pensar en este objeto como un marco de datos separado para cada grupo. Cada grupo se ha creado en función de las categorías en una columna agrupada (se crearán 4 grupos 'Black', 'Blue', 'Red', 'Yellow' a partir de la columna 'colour' del marco de datos en nuestro caso).

Un objeto GroupBy almacena los datos de los grupos individuales en forma de pares de valores clave como en el diccionario. Para conocer los nombres de los grupos, podemos usar el atributo 'keys' o usar el atributo 'groups' del objeto GroupBy.

```
#  comprobemos los nombres de los grupos

data_gby.groups

{'Black': [1, 5, 9, 13],
      'Blue': [2, 6, 10, 14, 16],
      'Red': [3, 7, 11, 15, 17],
      'Yellow': [0, 4, 8, 12]}
```

Para mayor claridad sobre los Grupos y su contenido, podemos ejecutar un bucle e imprimir los pares de valores clave.

```
### 'key' es el nombre del grupo y 'value' son las
filas segmentadas del DataFrame original.
```

```
for key, value in data_gby:

   print('GroupName: ',key)

   print(value)

   print('----------------------------------------
')
```

```
GroupName:  Black
  colour   sales  transactions product                                  1   Black
150000         150  type A
5   Black  145000         120  type A
9   Black  200000         110  type B                                  13  Black
150000         300  type B
----------------------------------------
GroupName:  Blue
  colour   sales  transactions product                                  2    Blue
80000          820  type A
6    Blue  120000          70  type A
10   Blue  160000         130  type B
14   Blue  142000         150  type B
16   Blue  400000         230  type B
----------------------------------------
GroupName:  Red
  colour   sales  transactions product
3     Red   90000         920  type A
7     Red  300000         250  type A
11    Red   90000         860  type B
15    Red  130000         170  type B
17    Red  350000         280  type B
----------------------------------------
GroupName:  Yellow
   colour   sales  transactions product
0   Yellow  100000         100  type A
4   Yellow  200000         230  type A
8   Yellow  250000         250  type A
12  Yellow   90100         980  type B
----------------------------------------
```

APPLY: Aplicar alguna función en el Objeto.

El paso Aplicar se puede realizar de tres maneras: *Agregación, Transformación y Filtrado.* Todos tenemos una buena cantidad de experiencia en el uso de Agregación con objetos GroupBy, pero es posible que la mayoría de nosotros no tengamos la misma experiencia con la Transformación y el Filtrado. Aquí, discutiremos los tres con un enfoque especial en la Transformación.

1. AGREGACIÓN:

Supongo que ya nos sentimos cómodos aplicando las funciones de agregación con el objeto GroupBy, por lo tanto, comenzaré con algunas características interesantes de esta función.

Agregando en los Grupos creados por múltiples columnas:

Al elegir varias columnas para crear el grupo, aumentamos la granularidad de la agregación. Por ejemplo, al dividir creamos 4 grupos basados en la columna 'colours', que tiene 4 categorías de colores, por lo que teníamos 4 grupos. Ahora, si incluye la columna 'product', que tiene 2 categorías ('type A' y 'type B'), junto con la columna 'colour', entonces tendremos un total de 8 categorías (por ejemplo, 'type A-Blue', 'type A-Blanck', …) en total (4 x 2). Esto sería más claro a partir del código mencionado a continuación.

```
data_prod_colour_index=data_sales.groupby(['product',
'colour'], as_index=True).sum()      # Note: as_index
= True

data_prod_colour_index
```

		sales	transactions
product	colour		
type A	Black	295000	270
	Blue	200000	890
	Red	390000	1170
	Yellow	550000	580
type B	Black	350000	410
	Blue	702000	510
	Red	570000	1310
	Yellow	90100	980

El código anterior usaba la función de agregación como sum(), por lo que obtenemos la suma de las ventas y las transacciones al nivel de granularidad definido por la combinación de las columnas 'product' y 'colour'.

Cabe señalar que hemos utilizado el parámetro 'as_index=True', por lo que podemos ver la columna 'product' y 'colour' como índice. Por el contrario, si tomamos el mismo parámetro como Falso, en nuestra salida no obtendremos las columnas 'product' y 'colour' como índice sino como columnas.

```
data_prod_colour_Noindex=data_sales.groupby(['product
', 'colour'],as_index=False).sum()


data_prod_colour_Noindex
```

	product	colour	sales	transactions
0	type A	Black	295000	270
1	type A	Blue	200000	890
2	type A	Red	390000	1170
3	type A	Yellow	550000	580
4	type B	Black	350000	410
5	type B	Blue	702000	510
6	type B	Red	570000	1310
7	type B	Yellow	90100	980

Agregación personalizada agrupada por múltiples columnas

En el ejemplo anterior, usamos un solo tipo de función de agregación para todas las columnas; sin embargo, si queremos agregar diferentes columnas con diferentes funciones de agregación, podemos usar la funcionalidad de agregación personalizada de la función de agregación. Para hacer esto, podemos pasar el diccionario a la función de agregación indicando el nombre de la columna como 'key' y el nombre de la función como 'value'. Curiosamente, también podemos pasar las múltiples funciones de agregación a una columna. Veamos un código de ejemplo a continuación para mayor claridad.

```
data_sales.groupby(['product','colour'],
as_index=True).agg({'sales':np.sum,
'transactions':[np.median,'count']})
```

		sales	transactions	
		sum	median	count
product	colour			
type A	Black	295000	135.0	2
	Blue	200000	445.0	2
	Red	390000	585.0	2
	Yellow	550000	230.0	3
type B	Black	350000	205.0	2
	Blue	702000	150.0	3
	Red	570000	280.0	3
	Yellow	90100	980.0	1

Ejemplo:

Hace unos días, uno de mis amigos me pidió que calculara la volatilidad de diferentes grupos de datos. Le sugerí que usara el 'coeficiente de variación' como medida de volatilidad. ¿Por qué solo 'coeficiente de variación' pero por qué no 'varianza'?. Se puede usar un tipo de ejemplo similar para mostrar la aplicación de la función de agregación en los negocios. Solo como ejemplo, usaremos los datos de ventas antes mencionados para ilustrar su aplicación.

En este ejemplo, estamos tratando de encontrar qué producto y su combinación de colores tienen la variación más baja. Hemos hecho esto agrupando el marco de datos en función de las columnas de producto y color y luego calculando el coeficiente de variación para cada grupo. El siguiente código lo hará más claro.

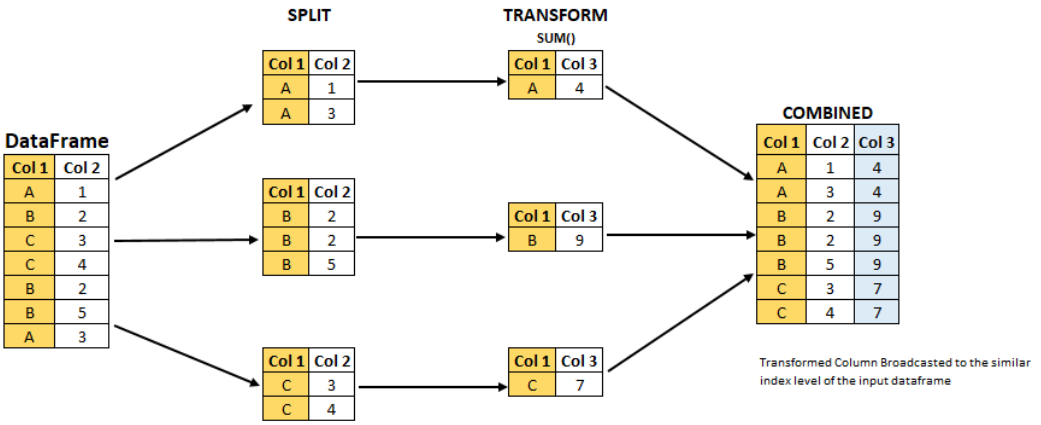
```
## Definiendo la funcion

def coef_de_variacion(x):

 cc_vv = x.std()/x.mean()
```

```
 return(cc_vv)


## utilizar la función definida anteriormente en la agregación

cvn = data_sales.groupby(['product', 'colour'])['sales'].agg([coef_de_variacion, np.mean, np.std])

print(cvn) # El grupo "Type A - Blanck" tiene la volatilidad más baja
```

```
                    coef_de_variacion       mean          std
product colour
type A  Black                0.023970  147500.000000    3535.533906
        Blue                 0.282843  100000.000000   28284.271247
        Red                  0.761500  195000.000000  148492.424049
        Yellow               0.416598  183333.333333   76376.261583
type B  Black                0.202031  175000.000000   35355.339059
        Blue                 0.615563  234000.000000  144041.660640
        Red                  0.736842  190000.000000  140000.000000
        Yellow                    NaN   90100.000000          NaN
```

2. TRANSFORMACIÓN:

La función de transformación tiene una gran utilidad potencial en la ingeniería de características. Es una función/método utilizado junto con el objeto GroupBy. Es un poco contrario a la intuición, por lo tanto, un poco difícil de entender. Si alguien ha utilizado la función LOD (Fixed) de Tableau, será más fácil para ellos comprender la función de transformación. La siguiente figura ilustra Split-Apply-Combine usando la función de transformación.



Durante la agregación, vimos que la forma del marco de datos de entrada se redujo (se redujo el número de filas); sin embargo, se debe tener en cuenta que al usar el método de transformación, el marco de datos de salida resultante tiene el mismo número de filas en la salida que en la entrada. La salida retuvo la longitud del marco de datos y sucedió en dos etapas. En primer lugar, en la etapa de aplicación, se aplicó la función de transformación (sum(); como se muestra en la Figura anterior), y en esta etapa se redujo el número de filas. En segundo lugar, en la etapa de combinación, el resultado de la etapa Aplicar se transmitió al nivel original de granularidad, produciendo así el marco de datos con la misma longitud que el marco de datos en la etapa de entrada.

Para que quede más claro, podemos usar un código de ejemplo usando el mismo conjunto de datos ficticios que hemos estado usando en este artículo.

```
### Aplicar la función de transformación (desviación estándar:std()) sobre la columna de ventas agrupadas por columna de producto.

### Hay que tener en cuenta que la salida tiene el mismo número de filas que en la entrada
```

```
data_sales['sales_product_std'] =
data_sales.groupby('product',as_index=True)['sales'].
transform('std')


data_sales.loc[:,['colour','product','sales','transac
tions', 'sales_product_std']]
```

	colour	product	sales	transactions	sales_product_std
0	Yellow	type A	100000	100	75929.426297
1	Black	type A	150000	150	75929.426297
2	Blue	type A	80000	820	75929.426297
3	Red	type A	90000	920	75929.426297
4	Yellow	type A	200000	230	75929.426297
5	Black	type A	145000	120	75929.426297
6	Blue	type A	120000	70	75929.426297
7	Red	type A	300000	250	75929.426297
8	Yellow	type A	250000	250	75929.426297
9	Black	type B	200000	110	110783.301991
10	Blue	type B	160000	130	110783.301991
11	Red	type B	90000	860	110783.301991
12	Yellow	type B	90100	980	110783.301991
13	Black	type B	150000	300	110783.301991
14	Blue	type B	142000	150	110783.301991
15	Red	type B	130000	170	110783.301991
16	Blue	type B	400000	230	110783.301991
17	Red	type B	350000	280	110783.301991

Ya que ahora tenemos cierta comprensión de la función de transformación, ahora hablemos de su utilidad en la limpieza de datos y la ingeniería de características.

Podemos definir una función personalizada y usarla para transformar la columna. Por ejemplo, podemos usar un ejemplo común de estandarización de una columna basada en la categoría de grupo a través de la función de transformación. *Cabe señalar que esta estandarización no se aplica directamente a toda la columna, sino que se aplica a la columna según el grupo (la media es la media del grupo y la desviación estándar es la desviación estándar del grupo)*. El siguiente ejemplo arrojará más luz sobre este concepto.

```
## Usando la función personalizada para transformar.
Estandarización de la columna 'sales' agrupada por la
columna 'colour'.


data_sales['sales_stdzed_colour'] =
data_sales.groupby('colour')['sales'].transform(lambd
a x: (x-x.mean())/x.std())


print(data_sales.loc[:,['colour', 'sales',
'sales_stdzed_colour']])
```

	colour	sales	sales_stdzed_colour
0	Yellow	100000	-0.770946
1	Black	150000	-0.433682
2	Blue	80000	-0.794706
3	Red	90000	-0.824083
4	Yellow	200000	0.513429
5	Black	145000	-0.626430
6	Blue	120000	-0.478090
7	Red	300000	0.872558
8	Yellow	250000	1.155617
9	Black	200000	1.493795
10	Blue	160000	-0.161474
11	Red	90000	-0.824083

12	Yellow	90100	-0.898099
13	Black	150000	-0.433682
14	Blue	142000	-0.303951
15	Red	130000	-0.500913
16	Blue	400000	1.738221
17	Red	350000	1.276520

En el código anterior, hemos usado la función lambda y dentro de la función lambda hemos usado los dos métodos media y desviación estándar (std) en cada fila del marco de datos. *Puede surgir una confusión a partir de aquí de '¿cómo una función de transformación está haciendo el cálculo por filas, cuando está destinado a hacer un cálculo por grupos?'.* Toda la historia detrás de la escena se explica en el código a continuación.

```
## Cálculo de la media a nivel de grupo y su difusión
en el marco de datos original.


data_sales['X.mean'] =
data_sales.groupby('colour')['sales'].transform('mean
')

## Cálculo de la desviación estándar a nivel de grupo
y su difusión en el marco de datos original.


data_sales['X.std'] =
data_sales.groupby('colour')['sales'].transform('std'
)

## Estandarización simple por filas basada en las
columnas X, X.mean , y X.std


data_sales['simple_stdzed'] = (data_sales['sales']-
data_sales['X.mean'])/data_sales['X.std']
```

```
## Imprime tanto la columna transformada como la
columna con cálculo simple y observa la diferencia
entre ambas.


## Ambas columnas 'simple_stdzed', y
'sales_stdzed_colour' son exactamente iguales.


## La media y la desviación estándar se han calculado
en función de los grupos, por lo que podemos ver que
los valores (X.media, X.std) se repiten para los
mismos colores.


print(data_sales.loc[:,['colour',
'sales','X.mean','X.std','simple_stdzed','sales_stdze
d_colour']])
```

Usted puede copiar el código anterior para checar el uso de la operación de transformación para que sea más clara.

3. FILTRACIÓN:

Como es evidente por el propio nombre, se utiliza para filtrar los grupos de los marcos de datos. El código mencionado a continuación ilustra la operación.

Si queremos filtrar el marco de datos de forma que contenga sólo aquellos colores que tengan un número medio de transacciones mayor que la media de algún otro color, podemos hacer lo siguiente:

```
grouped = data_sales.groupby('colour')
```

```
Blue_avg_transaction=
grouped['transactions','sales'].mean().loc['Blue','tr
ansactions']

Black_avg_transaction=
grouped['transactions','sales'].mean().loc['Black','t
ransactions']

Yellow_avg_transaction=
grouped['transactions','sales'].mean().loc['Yellow','
transactions']

Red_avg_transaction=
grouped['transactions','sales'].mean().loc['Red','tra
nsactions']


print('Blue_avg_transaction: ', Blue_avg_transaction)

print('Black_avg_transaction:',
Black_avg_transaction)

print('Yellow_avg_transaction: ',
Yellow_avg_transaction)

print('Red_avg_transaction:', Red_avg_transaction)
```

```
Blue_avg_transaction:  280.0
Black_avg_transaction: 170.0
Yellow_avg_transaction:  390.0
Red_avg_transaction: 496.0
```

```
## El resultado muestra que
```

```
filt_df = grouped.filter(lambda x:
x['transactions'].mean() > Black_avg_transaction)


print(filt_df.iloc[:,[0,2]])
```

	colour	transactions
0	Yellow	100
2	Blue	820
3	Red	920
4	Yellow	230
6	Blue	70
7	Red	250
8	Yellow	250
10	Blue	130
11	Red	860
12	Yellow	980
14	Blue	150
15	Red	170
16	Blue	230
17	Red	280

El filtrado es una operación fácil de entender y con esta sección APPLY de SPLIT-APPLY-COMBINE llega a su fin. Ahora, pasaremos a la última parte, que es COMBINE.

COMBINE: Combina el resultado para obtener un nuevo Objeto.

Durante las discusiones anteriores, la sección de la cosechadora ya ha sido cubierta; sin embargo, hay un punto importante con respecto a esto que me gustaría compartir.

```
## Función de agregación mean() aplicada a una sola columna "sales". Tenga en cuenta también as_index=True.

d1 = data_sales.groupby('colour', as_index=True)['sales'].mean()

type(d1) ## La salida es la Serie Pandas

print(d1)
```

```
colour
Black     161250.0
Blue      180400.0
Red       192000.0
Yellow    160025.0
Name: sales, dtype: float64
```

```
## Función de agregación mean() aplicada sólo a dos columnas

d2 = data_sales.groupby('colour',as_index=True).agg({'sales':np.mean, 'transactions': np.sum})

type(d2)     ## La salida es DataFrame Pandas

print(d2)
```

```
            sales  transactions
colour
Black   161250.0           680
Blue    180400.0          1400
```

```
Red     192000.0          2480
Yellow  160025.0          1560
```

```
## La función de agregación mean() se aplica a una sola columna 'sales', pero ahora el resultado es un dataframe con el parámetro as_index=True

d3 = data_sales.groupby('colour', as_index=False)['sales'].mean()

type(d3)      ## La salida es DataFrame Pandas

print(d3)
```

```
   colour     sales
0   Black  161250.0
1    Blue  180400.0
2     Red  192000.0
3  Yellow  160025.0
```

La agregación no siempre conduce a la creación del marco de datos. Depende principalmente del parámetro 'as_index', si el valor de este parámetro es 'Verdadero', entonces depende del número de columnas en las que estamos aplicando la función de agregación.

Aplicando Quantiles y Buckets

Un **cuantil** es aquel punto que divide la función de distribución de una variable aleatoria en *intervalos regulares*.

Por tanto, no es más que una técnica estadística para separar los datos de una distribución. Eso sí, debe cumplirse que los grupos sean iguales. Por eso, existen diversos tipos de cuantil, como veremos más adelante, en función del número de particiones que hacen.

Cuantiles más frecuentes

La mayoría de ellos son de uso habitual para poder analizar de forma detallada la distribución de los datos. Además, otra de sus utilidades es separar los datos en grupos, pudiendo elegir los más altos o los más bajos.

- **Cuartil**: Separa los valores en cuatro grupos iguales y existen tres cuartiles. Es el más frecuente. El cuartil uno (Q1) son los datos menores y el tres (Q3) los mayores. Por otro lado, el cuartil dos (Q2) se corresponde con la mediana (Me) que es un estadístico de posición que divide la distribución de los datos a la mitad. Los valores del cuantil serían 0.25 (Q1), 0.5(Q2) y 0.75 (Q3).
- **Quintil**: Similar al anterior, es menos frecuente y divide los datos en cinco partes iguales. Por tanto, hay cuatro quintiles. Los valores del cuantil en este caso serían 0.20, 0.40, 0.60, 0.80.
- **Decil**: En este caso se dividen en diez partes y, por tanto, hay nueve deciles. Una vez más, este tampoco es demasiado frecuente. Sus valores serían de 0.1 a 0.9.
- **Percentiles**: Estamos ante una variante en que la distribución se divide en cien partes iguales. Puede ser de interés para muestras muy numerosas. Sus valores van de 0.01 a 0.99.

En Py

```
statistics.quantiles(data, *, n=4, method='exclusive')
```

Esto divide la data en **n intervalos** continuos equiprobables. Retorna una lista de **n - 1** límites que delimitan los intervalos (cuantiles). Para que los resultados sean significativos, el número de observaciones en la muestra data debe ser mayor que n. Si no hay al menos dos observaciones se lanza una excepción StatisticsError.

Los límites de los intervalos se interpolan linealmente a partir de los dos valores más cercanos de la muestra. Por ejemplo, si un límite es un tercio

de la distancia entre los valores 100 y 112 de la muestra, el límite será 104.

El argumento method indica el método que se utilizará para calcular los cuantiles y se puede modificar para especificar si se deben incluir o excluir valores de data extremos, altos y bajos, de la población.

El valor predeterminado para method es «exclusive» y es aplicable a los datos muestreados de una población que puede tener valores más extremos que los encontrados en las muestras. La proporción de la población que se encuentra por debajo del i-ésimo valor de m valores ordenados se calcula mediante la fórmula i / (m + 1). Por ejemplo, asumiendo que hay 9 valores en la muestra, este método los ordena y los asocia con los siguientes percentiles: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

Si se usa «inclusive» como valor para el parámetro method, se asume que los datos corresponden a una población completa o que los valores extremos de la población están representados en la muestra. El valor mínimo de data se considera entonces como percentil 0 y el máximo como percentil 100. La proporción de la población que se encuentra por debajo del i-ésimo valor de m valores ordenados se calcula mediante la fórmula (i - 1) / (m - 1). Suponiendo que tenemos 11 valores en la muestra, este método los ordena y los asocia con los siguientes percentiles: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80 %, 90%, 100%.

```
# Puntos de corte de los deciles para los datos
muestreados empíricamente

data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92,

             110,100, 75, 105, 103, 109, 76, 119, 99, 91,

103, 129, 106, 101, 84, 111, 74, 87, 86, 103,

103, 106, 86, 111, 75, 87, 102, 121, 111, 88,
```

```
89, 101, 106, 95,103, 107, 101, 81, 109, 104]

[round(q, 1) for q in quantiles(data, n=10)]

[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8,
111.0]
```

Ordenar por cubos en Python

Bucket Sort es un algoritmo de tipo de comparación que asigna elementos de una lista que queremos ordenar en Buckets o Bins. A continuación, se ordena el contenido de estos depósitos, normalmente con otro algoritmo. Después de ordenar, el contenido de los cubos se agrega, formando una colección ordenada.

La clasificación por cubos se puede considerar como un enfoque de dispersión, orden y recopilación para ordenar una lista, debido al hecho de que los elementos primero se dispersan en cubos, se ordenan dentro de ellos y finalmente se reúnen en una nueva lista ordenada.

Implementaremos Bucket Sort en Python y analizaremos su complejidad de tiempo.

¿Cómo funciona la clasificación por cubos?

Antes de pasar a su implementación exacta, repasemos los pasos del algoritmo:

- Configure una lista de depósitos vacíos. Se inicializa un depósito para cada elemento de la matriz.
- Repita la lista de deseos e inserte elementos de la matriz. El lugar donde se inserta cada elemento depende de la lista de entrada y

del elemento más grande de la misma. Podemos terminar con 0...n elementos en cada cubo. Esto se desarrollará en la presentación visual del algoritmo.

- Clasifica cada balde que no esté vacío. Puede hacer esto con cualquier algoritmo de clasificación. Dado que estamos trabajando con un conjunto de datos pequeño, cada depósito no tendrá muchos elementos, por lo que Insertion Sort funciona de maravilla para nosotros aquí.
- Visite los cubos en orden. Una vez que se ordena el contenido de cada depósito, cuando se concatenan, producirán una lista en la que los elementos se organizan según sus criterios.

Echemos un vistazo a la presentación visual de cómo funciona el algoritmo. Por ejemplo, supongamos que esta es la lista de entrada.

El elemento más grande es 1.2, y la longitud de la lista es 6. Usando estos dos, averiguaremos el óptimo size de cada cubo. Obtendremos este número dividiendo el elemento más grande por la longitud de la lista. En nuestro caso, es 1.2/6 el cual es 0.2.

Dividiendo el valor del elemento con este size, obtendremos un índice para el depósito respectivo de cada elemento. Ahora crearemos depósitos vacíos. Tendremos la misma cantidad de depósitos que los elementos de nuestra lista.

Insertamos los elementos en sus respectivos cubos. Teniendo en cuenta el primer elemento: 1.2/0.2 = 6, el índice de su respectivo segmento es 6. Si este resultado es mayor o igual a la longitud de la lista, simplemente restamos 1 y encajaría perfectamente en la lista. Esto solo sucede con el número más grande, ya que obtuvimos el size dividiendo el elemento más grande por la longitud.

Colocaremos este elemento en el depósito con el índice de 5. Asimismo, el siguiente elemento se indexará a 0.22/0.2 = 1.1. Dado que este es un número decimal, lo nivelaremos. Esto se redondea a 1, y nuestro elemento se coloca en el segundo depósito. Este proceso se repite hasta que hayamos colocado el último elemento en su respectivo depósito. Nuestros cubos ahora se ven algo así como.

Ahora, ordenaremos el contenido de cada cubo que no esté vacío. Usaremos el ordenamiento por inserción, ya que está invicto con listas pequeñas como esta. Después de la ordenación por inserción, los depósitos se ven así. Ahora, es solo cuestión de atravesar los depósitos que no están vacíos y concatenar los elementos en una lista. Están ordenados y listos para usar.

Implementación de clasificación de cubos en Python

Con eso fuera del camino, sigamos adelante e implementemos el algoritmo en Python. Empecemos con el bucket_sort() función en sí misma:

```
def bucket_sort(input_list):

   # Encuentra el valor máximo de la lista y utiliza la
longitud de la lista para determinar qué valor de la lista
va a cada cubo

   max_value = max(input_list)

   size = max_value/len(input_list)



   # Crear n cubos vacíos donde n es igual a la longitud
de la lista de entrada

   buckets_list= []
```

```
    for x in range(len(input_list)):
        buckets_list.append([])



    # Poner los elementos de la lista en diferentes cubos
en función del tamaño
    for i in range(len(input_list)):
        j = int (input_list[i] / size)
        if j != len (input_list):
            buckets_list[j].append(input_list[i])
        else:
            buckets_list[len(input_list) -
1].append(input_list[i])



    # Ordenar los elementos dentro de los cubos mediante el
orden por inserción
    for z in range(len(input_list)):
        insertion_sort(buckets_list[z])


    # Concatenar cubos con elementos ordenados en una sola
lista
    final_output = []
    for x in range(len (input_list)):
        final_output = final_output + buckets_list[x]
```

```
    return final_output
```

La implementación es bastante sencilla. Hemos calculado el size del parámetro. Luego, instanciamos una lista de depósitos vacíos e insertamos elementos basados en su valor y el size de cada cubo.

Una vez insertado, llamamos insertion_sort() en cada uno de los cubos:

```
def insertion_sort(bucket):
    for i in range (1, len (bucket)):
        var = bucket[i]
        j = i - 1
        while (j >= 0 and var < bucket[j]):
            bucket[j + 1] = bucket[j]
            j = j - 1
        bucket[j + 1] = var
```

Y con eso en su lugar, completamos una lista y realicemos una ordenación de cubos en ella:

```
def main():
    input_list = [1.20, 0.22, 0.43, 0.36,0.39,0.27]
    print('ORIGINAL LIST:')
    print(input_list)
    sorted_list = bucket_sort(input_list)
    print('SORTED LIST:')
```

```
   print(sorted_list)
Original list: [1.2, 0.22, 0.43, 0.36, 0.39, 0.27]
Sorted list: [0.22, 0.27, 0.36, 0.39, 0.43, 1.2]
```

Llenado de datos

La imputación de valores perdidos se puede dividir aproximadamente en tres tipos:

- Reemplazar los valores perdidos. El reemplazo se completa con la similitud de los datos que no faltan en los datos, y su idea central es encontrar las características comunes del mismo grupo.
- Ajuste los valores perdidos. El ajuste se rellena con otro modelado de características
- variable virtual. Las variables ficticias son nuevas variables derivadas en lugar de valores perdidos.

Nota: a veces se opta por eliminar filas que contienen datos faltantes, sin embargo, no suele ser recomendable.

Se estudiará más adelante el tratamiento de estas bases con técnicas más avanzadas.

Construcción de tablas pivote

Las tablas dinámicas son una muy buena forma para analizar datos. Con un clic, se puede profundizar en los detalles granulares sobre una determinada categoría de producto, o alejarme y obtener una descripción general de alto nivel de los datos disponibles.

Las tablas dinámicas ofrecen mucha flexibilidad como científico de datos. Los usuarios de Excel están íntimamente familiarizados con estas tablas dinámicas. Son la característica más utilizada de Excel , ¡y es fácil ver por qué! ¿Pero sabías que puedes construir estas tablas dinámicas usando **Pandas** en Python?

```
# pandas.DataFrame.pivot

DataFrame.pivot(index=None, columns=None, values=None)
```

Devuelve el DataFrame remodelado organizado por valores de índice/columna dados.

Remodele los datos (produzca una tabla "pivote") en función de los valores de las columnas. Utiliza valores únicos de índices / columnas especificados para formar ejes del DataFrame resultante. Esta función no admite la agregación de datos, varios valores darán como resultado un índice múltiple en las columnas. Consulte la Guía del usuario para obtener más información sobre la remodelación.

Parameters:

- **index:** str u object o una lista de str, opcional
  Columna a usar para hacer el índice del nuevo marco. Si no hay, usa el índice existente.
- **columns:** str u objeto o una lista de str
  Columna a usar para hacer las columnas del nuevo marco.
- **values:** str, object o una lista de los anteriores, opcional

Columna(s) a usar para completar los valores del nuevo marco. Si no se especifica, se utilizarán todas las columnas restantes y el resultado tendrá columnas indexadas jerárquicamente.

Returns: DataFrames

Devuelve DataFrame reformado.

Raises: ValueError:

Cuando hay algún índice, combinaciones de columnas con múltiples valores. DataFrame.pivot_table cuando necesite agregar.

Aquí hay unos ejemplos de cómo trabajar con un conjunto de datos.

```
import pandas as pd


df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two',
'two','two'],'bar': ['A', 'B', 'C', 'A', 'B', 'C'],'baz':
[1, 2, 3, 4, 5, 6],'zoo': ['x', 'y', 'z', 'q', 'w', 't']})

df
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo', columns='bar', values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

```
df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
```

	baz			zoo		
bar	A	B	C	A	B	C
foo						
one	1	2	3	x	y	z
two	4	5	6	q	w	t

También puede asignar una lista de nombres de columna o una lista de nombres de índice.

```
df = pd.DataFrame({"lev1": [1, 1, 1, 2, 2, 2], "lev2": [1, 1, 2, 1, 1, 2], "lev3": [1, 2, 1, 2, 1, 2], "lev4": [1, 2, 3, 4, 5, 6], "values": [0, 1, 2, 3, 4, 5]})

print(df.pivot(index="lev1", columns=["lev2", "lev3"],values="values"))
```

lev2	1		2	
lev3	1	2	1	2
lev1				
1	0.0	1.0	2.0	NaN
2	4.0	3.0	NaN	5.0

```
df.pivot(index=["lev1", "lev2"], columns=["lev3"],values="values")
```

	lev3	1	2
lev1	lev2		
1	1	0.0	1.0
	2	2.0	NaN
2	1	4.0	3.0
	2	NaN	5.0

Referencias

[1] Datos agrupados
https://www.ibm.com/docs/es/spss-statistics/SaaS?topic=chart-binning-grouping-data-values

[2] Función separar - aplicar - combinar
https://medium.com/analytics-vidhya/split-apply-combine-strategy-for-data-mining-4fd6e2a0cc99

[3] Conceptos estadísticos
https://docs.python.org/es/3/library/statistics.html

[4] Tablas pivote
https://ichi.pro/es/aqui-se-explica-como-construir-una-tabla-dinamica-usando-pandas-en-python-112177747736514