**¿Cómo lidiar con datos faltantes en tus DataFrames?**

Es muy común que nuestros DataFrames presenten datos faltantes, antes de empezar a procesar nuestros DataFrames veamos un poco en qué consisten los objetos **NaN** (Not a Number).

Importemos las librerias Pandas y Numpy para esto:

**import** numpy **as** np

**import** pandas **as** pd

Un número que no está definido usualmente se representa con el siguiente objeto:

np.nan

> nan

¡Este objeto tiene propiedades matemáticas! Al sumar un número, obtenemos como respuesta el mismo **NaN**.

np.nan + 0

> nan

np.nan > 0

> **False**

La versión 1.0 de pandas incluye un nuevo objeto **NA**, que es mucho más  
general pues, además de interactuar con números, también puede hacerlo con  
cadenas de texto u otras variables como las de tipo booleano. Si quieres que  
esta nueva definición este incluida entre tus cálculos usa:

pd.options.mode.use\_inf\_as\_na = True

Al sumar **NA** a una cadena de texto, obtengo el mismo **NA**:

pd.NA +'Hola mundo'

> <NA>

pd.NA | **False**

> <NA>

A continuación, vamos a crear un DataFrame

df = pd.DataFrame(np.arange(0, 15).reshape(5, 3), columns=['a', 'b', 'c'])

df

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b | c |
| 0 | 0 | 1 | 2 |
| 1 | 3 | 4 | 5 |
| 2 | 6 | 7 | 8 |
| 3 | 9 | 10 | 11 |
| 4 | 12 | 13 | 14 |

Y vamos a añadir algunas variables no definidas:

df['d'] = np.nan

df['e'] = np.arange(15, 20)

df.loc[5,:] = pd.NA

df.loc[4,'a'] = pd.NA

df.loc[0,'d'] = 1

df.loc[5,'d'] = 10

df

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0.0 | 1.0 | 2.0 | 1 | 15.0 |
| 1 | 3.0 | 4.0 | 5.0 | nan | 16.0 |
| 2 | 6.0 | 7.0 | 8.0 | nan | 17.0 |
| 3 | 9.0 | 10.0 | 11.0 | nan | 18.0 |
| 4 | <NA> | 13.0 | 14.0 | nan | 19.0 |
| 5 | <NA> | <NA> | <NA> | 10 | <NA> |

Para reconocer cuando un objeto es nulo simplmente usamos:

df.isnull()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 | 0 |
| 5 | 1 | 1 | 1 | 0 | 1 |

En dónde todas nuestras variables no definidas fueron marcadas con TRUE,  
df.isna() tambien cumple esta función.

Conocer el número de variables nulas por columna puede hacerse juntando el  
comando anterior con la funcion de suma:

df.isnull().sum()

|  |  |
| --- | --- |
|  |  |
| a | 2 |
| b | 1 |
| c | 1 |
| d | 4 |
| e | 1 |

dtype: int64

Si lo que nos interesa es conocer el número de filas con elementos nulos, basta  
con usar axis=1:

df.notnull().sum(axis=1)

|  |  |
| --- | --- |
|  |  |
| 0 | 5 |
| 1 | 4 |
| 2 | 4 |
| 3 | 4 |
| 4 | 3 |
| 5 | 1 |

dtype: int64

O todos los elementos nulos de nuestro DataFrame:

df.size-df.isnull().sum().sum()

21

Reconocer estos elementos nos puede ayudar a filtrar en nuestro DataFrame, en  
este caso, me gustaría filtrar por las variables no nulas de la columna ‘a’:

df[df['a'].notnull()]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 1 | 2 | 1 | 15 |
| 1 | 3 | 4 | 5 | nan | 16 |
| 2 | 6 | 7 | 8 | nan | 17 |
| 3 | 9 | 10 | 11 | nan | 18 |

dropna es perfecto para elimnar rapidamente las filas con registros faltantes:

df.dropna()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 1 | 2 | 1 | 15 |

df[['a']].dropna()

|  |  |
| --- | --- |
|  | a |
| 0 | 0 |
| 1 | 3 |
| 2 | 6 |
| 3 | 9 |

Ya que hemos visto cómo funcionan las variables nulas, veamos cómo lidiar con  
ellas. Usando la función fillna podremos reemplazarlas por el valor que  
querramos, en este caso 0.

df.fillna(0)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 1 | 2 | 1 | 15 |
| 1 | 3 | 4 | 5 | 0 | 16 |
| 2 | 6 | 7 | 8 | 0 | 17 |
| 3 | 9 | 10 | 11 | 0 | 18 |
| 4 | 0 | 13 | 14 | 0 | 19 |
| 5 | 0 | 0 | 0 | 10 | 0 |

Si quisieramos remplazar con el valor siguiente usamos  
method="ffill":

df.fillna(method="ffill")

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 1 | 2 | 1 | 15 |
| 1 | 3 | 4 | 5 | 1 | 16 |
| 2 | 6 | 7 | 8 | 1 | 17 |
| 3 | 9 | 10 | 11 | 1 | 18 |
| 4 | 9 | 13 | 14 | 1 | 19 |
| 5 | 9 | 13 | 14 | 10 | 19 |

Si quisieramos remplazar con el valor previo usamos  
method="bfill":

df.fillna(method="bfill")

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0.0 | 1.0 | 2.0 | 1 | 15.0 |
| 1 | 3.0 | 4.0 | 5.0 | 10 | 16.0 |
| 2 | 6.0 | 7.0 | 8.0 | 10 | 17.0 |
| 3 | 9.0 | 10.0 | 11.0 | 10 | 18.0 |
| 4 | <NA> | 13.0 | 14.0 | 10 | 19.0 |
| 5 | <NA> | <NA> | <NA> | 10 | <NA> |

El mismo ejercicio anterior se puede aplicar con las filas usando axis=1:

df.fillna(method="bfill",axis=1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 1 | 2 | 1 | 15.0 |
| 1 | 3 | 4 | 5 | 16 | 16.0 |
| 2 | 6 | 7 | 8 | 17 | 17.0 |
| 3 | 9 | 10 | 11 | 18 | 18.0 |
| 4 | 13 | 13 | 14 | 19 | 19.0 |
| 5 | 10 | 10 | 10 | 10 | <NA> |

Podemos usar también una serie para reemplazar los valores de una columna en específico, es importante que haya emparejamiento entre los índices:

fill = pd.Series([100, 101, 102])

fill

|  |  |
| --- | --- |
|  |  |
| 0 | 100 |
| 1 | 101 |
| 2 | 102 |

dtype: int64

df['d'] = df['d'].fillna(fill)

df['d']

|  |  |
| --- | --- |
|  |  |
| 0 | 1 |
| 1 | 101 |
| 2 | 102 |
| 3 | nan |
| 4 | nan |
| 5 | 10 |

Name: d, dtype: float64

df

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0.0 | 1.0 | 2.0 | 1 | 15.0 |
| 1 | 3.0 | 4.0 | 5.0 | 101 | 16.0 |
| 2 | 6.0 | 7.0 | 8.0 | 102 | 17.0 |
| 3 | 9.0 | 10.0 | 11.0 | nan | 18.0 |
| 4 | <NA> | 13.0 | 14.0 | nan | 19.0 |
| 5 | <NA> | <NA> | <NA> | 10 | <NA> |

Una de las formas más usadas para reemplazar datos es usar el promedio de las columnas, esto se hace con la función mean. O si se quiere un mejor estimador, usamos median.

df.fillna(df.median())

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| 0 | 0 | 1 | 2 | 1 | 15 |
| 1 | 3 | 4 | 5 | 101 | 16 |
| 2 | 6 | 7 | 8 | 102 | 17 |
| 3 | 9 | 10 | 11 | 55.5 | 18 |
| 4 | 4.5 | 13 | 14 | 55.5 | 19 |
| 5 | 4.5 | 7 | 8 | 10 | 17 |

Por último, Pandas también puede interpolar los valores faltantes calculando el valor que puede haber existido en el medio.

df\_d = pd.concat([df[['d']], df[['d']].interpolate()],axis=1)

df\_d.columns = ['d\_antes','d\_interpolado']

df\_d

|  |  |  |
| --- | --- | --- |
|  | d\_antes | d\_interpolado |
| 0 | 1 | 1 |
| 1 | 101 | 101 |
| 2 | 102 | 102 |
| 3 | nan | 71.3333 |
| 4 | nan | 40.6667 |
| 5 | 10 | 10 |