Pandas 2

- Otros métodos de consultas
- Limpieza y transformación del dataset

Contenidos del trimestre

Librería Pandas

- Consultas con query
- Limpieza y preparación de los datos:
  - Gestión de datos ausentes
  - Transformación de datos
  - Detección y filtrado de valores atípicos
  - Discretización de variables
- Combinación y fusión de datasets
- Agregación y operaciones con grupos
- Visualización de datos directamente con pandas. Otras librerías de visualización

Otros modelos y librerías de ML y análisis de datos

- Análisis probabilístico: Clasificación Bayesiana (NaiveBayes con Scikit-Learn)
- Análisis de Varianza (ANOVA)
- Algoritmos no supervisados (clustering)
- Regresión logística, regresión polinómica
- Redes neuronales

Método query

La clase dataFrame tiene un método query para realizar consultas más directas mediante un string pasado por parámetro.

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.query.html#pandas-dataframe-query

Sintaxis básica: dataFrame.query('expresión')

La expresión va entre comillas simples. La expresión puede contener:

- El nombre de una o varias columnas
- Valores literales. Si los valores son cadenas de caracteres estos deben ir entre comillas dobles.
- Operadores aritméticos: + - * / ** % //
- Operadores de comparación: < > == !=
- Operadores booleanos: | (or), & (and), and ~ (not), además del uso directo de and, or, not.

Si después de la consulta deseamos solamente varias columnas, indexamos de manera normal. -> df.query(exp)[[columnas]]

Método query

Dado el dataset

	Number	City	Gender	Age	Income	Illness
0	1	Dallas	Male	41	40367.0	No
1	2	Dallas	Male	54	45084.0	No
2	3	Dallas	Male	42	52483.0	No
3	4	Dallas	Male	40	40941.0	No
4	5	Dallas	Male	46	50289.0	No

Ejemplos de consultas con query

```
# Búsqueda por un valor exacto de tipo string, requiere ""
dataset.query('Gender == "Male"')

# Búsqueda haciendo uso del índice y otra columna con operador and
dataset.query('index < 4 and City == "Dallass"')

# Búsqueda donde Edad está dentro de los valores de una tupla
dataset.query('Age in (41,42)')

# Búsqueda haciendo uso de una variable y f-strings
ciudad_buscada = "Dallas"
dataset.query(f'City == "{ciudad_buscada}"')
```

Método query

También se pueden aplicar algunas funciones en las expresiones

- df.query(‘columna.notnull()’) # Mostrar filas donde la columna no sea null
- df.query(‘columna.isnull()’) # Mostrar filas donde la columna es null
- df.query(‘columna.round(0) == 15’) # Redondeamos el valor consultado
- df.query(‘columna.abs() == 15’) # Valor absoluto del consultado

Operación aritmética entre 2 columnas

```
df.query(‘col1 + col2 > 5’)
```

Consultar a través de una lista

```
lista = [1, 2, 3]

df.query(‘columna in @lista’)
```

Limpieza y preparación de datos

Para realizar análisis y modelado de datos se necesita en primer lugar recolectar datos de diversas fuentes, unirlos, limpiarlos, transformarlos y reordenarlos. Normalmente estas tareas ocupan al analista de datos en torno al 80% del tiempo.

Estas tareas pueden hacerse mediante lenguajes de propósito general (Java, Python) o bien con herramientas de proceso de texto en Unix (sort, cut, sed, awk). Pero Pandas tiene herramientas de alto nivel, flexibles y rápidas para manipular y transformar los datos.

- Datos que faltan: filtrado, rellenado
- Transformación de datos: Eliminar duplicados, reemplazar valores, renombrar índices de eje, discretización
- Detección y filtrado de valores atípicos
- Calcular variables dummy o indicadoras
- Manipulación de cadenas de texto

Datos ausentes

En muchas aplicaciones de análisis de datos suele haber datos ausentes, faltantes o perdidos. A veces unimos varias fuentes de datos y algunos atributos pueden estar ausentes para determinadas filas. Imagina unir varias bases de datos con información de países (PIB, población, encuestas sobre bienestar, trabajo, salud), normalmente no vas a encontrar información de todos los países en todas las bases de datos.

En Pandas, todas las estadísticas descriptivas (sum, mean, max…) sobre objetos pandas (series, dataFrame) dejan fuera a los datos ausentes de forma predeterminada.

Pandas representa los ausentes, para los datos float64, como numpy.nan. Pero se refiere a ellos como NA (Not Availabe, no disponible). El valor None interno de Python también se trata como NA.

```
import pandas as pd
import numpy as np

serie = pd.Series(["manzana", np.nan, None, "pera"])
serie
```

```
0    manzana
1        NaN
2       None
3       pera
dtype: object
```

Datos ausentes

```
serie = pd.Series(["manzana", np.nan, None, "pera"])
serie
```

```
0    manzana
1        NaN
2       None
3       pera
dtype: object
```

Métodos a aplicar a una serie o dataframe:

- Comprobar si existen nulos, devuelve serie booleana.

  Se puede usar posteriormente any() u all()

```
serie.isna()
```

```
0    False
1     True
2     True
3    False
dtype: bool
```

- Comprobar si no existen nulos, devuelve serie booleana.

```
serie.notna()
```

```
0     True
1    False
2    False
3     True
dtype: bool
```

Estos métodos pueden usarse para indexar.

```
serie[serie.notna()]
```

```
0    manzana
3       pera
dtype: object
```

Filtrado de datos que faltan

Eliminar valores nulos: dropna(inplace=False) devuelve el objeto sin los valores nulos

```
serie[serie.notna()]
0    manzana
3       pera
dtype: object
```

```
serie.dropna()
0    manzana
3       pera
dtype: object
```

mismo resultado que con indexado booleano

En un dataFrame hay distintas maneras de eliminar los datos que faltan: Eliminar filas donde falta 1 valor, eliminar columnas donde falta 1 valor, eliminar filas o columnas si son todas nulas. Por defecto dropna() elimina cualquier fila que contiene 1 valor ausente. Podemos borrar solamente los nulos que faltan en un conjunto de columnas con parámetro subset=[]

df

	0	1	2
0	1.2	6.5	3.4
1	1.3	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

df.dropna()

	0	1	2
0	1.2	6.5	3.4

Filtrado de datos que faltan

	0	1	2	3
0	1.2	6.5	3.4	NaN
1	1.3	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

Parámetro how=”all” quitará solamente las filas que sean todas nulas

```
df.dropna(how="all")
```

	0	1	2	3
0	1.2	6.5	3.4	NaN
1	1.3	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

Para eliminar columnas usamos parámetro axis=”columns”

```
df.dropna(how="all", axis="columns")
```

	0	1	2
0	1.2	6.5	3.4
1	1.3	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Quedarnos solamente con las filas que tengan al menos N no ausentes, parámetro thresh=N

```
df.dropna(thresh=2)
```

	0	1	2	3
0	1.2	6.5	3.4	NaN
3	NaN	6.5	3.0	NaN

Rellenado de datos ausentes

Si eliminamos filas con valores ausentes estamos perdiendo información. A veces puede ser más conveniente rellenar los datos ausentes. Un método útil para la mayoría de casos es fillna()

DataFrame.fillna(X) rellena todos los ausentes por el valor X.

```
df.fillna(0)
```

	0	1	2	3
0	1.2	6.5	3.4	0.0
1	1.3	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	6.5	3.0	0.0

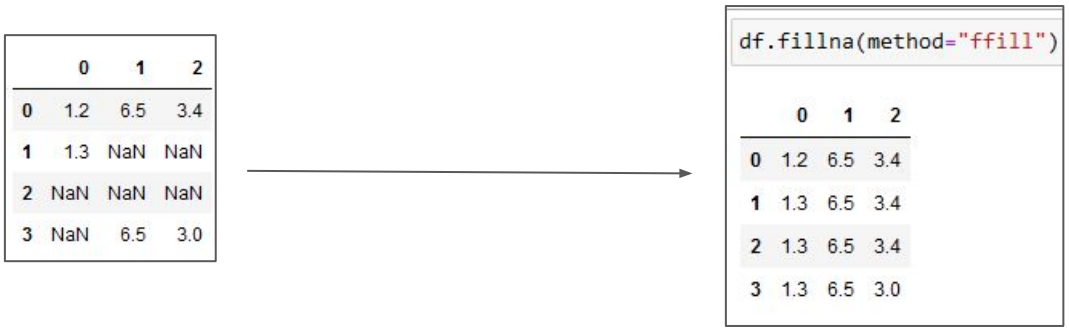
Podemos indicar 1 valor de relleno por cada columna con un diccionario

```
df.fillna({0:0.5, 1:1.2, 2:2.5 , 3:3})
```

	0	1	2	3
0	1.2	6.5	3.4	3.0
1	1.3	1.2	2.5	3.0
2	0.5	1.2	2.5	3.0
3	0.5	6.5	3.0	3.0

Rellenado de datos ausentes

También podemos rellenar los valores ausentes de 1 columna a partir del dato válido de la columna anterior (rellenado hacia adelante: ffill) o posterior (rellenado hacia atrás: bfill)



	0	1	2
0	1.2	6.5	3.4
1	1.3	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
df.fillna(method="ffill")
```

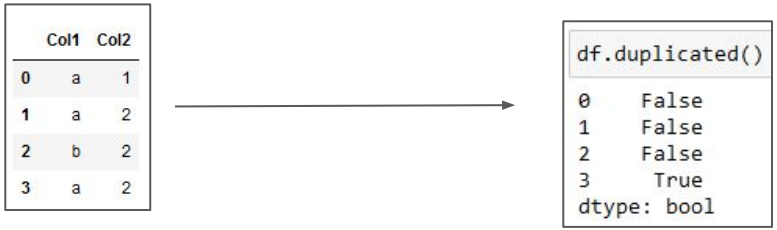
	0	1	2
0	1.2	6.5	3.4
1	1.3	6.5	3.4
2	1.3	6.5	3.4
3	1.3	6.5	3.0

También se puede limitar el rellenado a un número de transformaciones (limit=N) o rellenar por columnas en vez de por filas (axis=”columns”, por defecto es axis=”index”)

En valores numéricos puede ser interesante rellenar con la media o la mediana de la propia columna.

Eliminar duplicados

- duplicated() Devuelve una serie booleana indicando si cada fila es un duplicado de otra anterior.



	Col1	Col2
0	a	1
1	a	2
2	b	2
3	a	2

```
df.duplicated()

0    False
1    False
2    False
3     True
dtype: bool
```

- drop_duplicates() Devuelve un dataframe eliminando las filas donde duplicated es True. Ambos métodos tienen en cuenta todas las columnas. Como alternativa se puede indicar un subconjunto de columnas con el argumento subset=[“col1”, “col2”,..]. Drop conserva por defecto la primera combinación de valor en ser valorada, si pasamos como argumento keep=”last” será la última.

```
df.drop_duplicates()
```

	Col1	Col2
0	a	1
1	a	2
2	b	2

Transformación de datos mediante map

El método map de un objeto Series acepta una función o un diccionario que contiene un mapeado para hacer la transformación de valores.

Imagina que al siguiente dataframe queremos añadirle una columna para indicar el tipo de animal

	food	cantidad
0	bacon	4
1	pulled pork	3
2	corned beef	6
3	bacon	3
4	honey ham	5
5	nova lox	6

Para hacer el mapeo de datos creamos un diccionario y se lo pasamos a la función map

En el ejemplo hemos creado 1 columna, pero se puede hacer sobre la misma columna del map.

```
meat_to_animal = {"bacon": "pig","pulled pork": "pig","corned beef": "cow","honey ham": "pig","nova lox": "salmon
data["animal"] = data["food"].map(meat_to_animal)
data
```

	food	cantidad	animal
0	bacon	4	pig
1	pulled pork	3	pig
2	corned beef	6	cow
3	bacon	3	pig
4	honey ham	5	pig
5	nova lox	6	salmon

Reemplazar valores de una columna

Fillna es un caso especial de reemplazo de valores (reemplaza los nulos). Map también se puede utilizar para modificar un conjunto de valores. Pero replace es un método general para modificar de forma sencilla y flexible.

Podemos reemplazar varios valores de una vez ofreciendo 2 listas: replace([valores_presentes], [valores_nuevos])

Valores nuevos puede ser solamente 1 valor mientras que valores_presentes una lista, de esta forma reemplazará varios valores por 1 solo valor de una pasada.

Replace devuelve una serie con los valores modificados. Para modificar el DataFrame existente, argumento inplace=true

	food	cantidad
0	bacon	4
1	pulled pork	3
2	corned beef	6
3	bacon	3
4	honey ham	5
5	nova lox	6


```
data['food'].replace(['bacon', 'pulled pork'], 'pork', inplace=True)
data
```

	food	cantidad
0	pork	4
1	pork	3
2	corned beef	6
3	pork	3
4	honey ham	5
5	nova lox	6

Renombrar índices

Los dataFrames y las Series tienen un objeto index que identifica/nombra el índice de filas. Mientras que el dataFrame tiene un segundo objeto index que almacena las columnas. Ambos objetos se almacenan en sendos atributos del dataFrame llamados **index** y **columns** respectivamente.

Dado el dataFrame a partir del titanic.csv

```
import pandas as pd

df = pd.read_csv('titanic.csv')
```

```
# índice de filas
df.index
```

```
RangeIndex(start=0, stop=891, step=1)
```

```
# índice de columnas
df.columns
```

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

Renombrar índices

Al igual que los valores de una Serie, las etiquetas de cada eje se pueden transformar mediante funciones o una asignación directa.

Cambio directo mediante asignación de una lista de nombres:

```
nuevas_columnas=['id', 'sobrevive', 'clase', 'nombre', 'sexo', 'edad', 'hermanos', 'padres/hijos', 'ticket',
                 'tarifa', 'camarote', 'puerto']
df.columns = nuevas_columnas
df.head(3)
```

	id	sobrevive	clase	nombre	sexo	edad	hermanos	padres/hijos	ticket	tarifa	camarote	puerto
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S

Renombrar índices

Método rename para crear una versión transformada de un dataFrame sin modificar el original. Rename devuelve un nuevo dataframe. Puede modificar tanto las etiquetas de las filas (index) como los nombres de las columnas (columns). Admite funciones de tipo string como tittle (1ª letra a mayúscula, resto minúsculas), upper, lower.

```
df2 = df.rename(columns=str.upper)
df2.head(3)
```

	ID	SOBREVIVE	CLASE	NOMBRE	SEXO	EDAD	HERMANOS	PADRES/ HIJOS	TICKET	TARIFA	CAMAROTE	PUERTO
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S

Renombrar índices

Método rename también admite un diccionario para modificar los valores de las etiquetas de forma más detallada

```
df_nuevo = df.rename(columns={'id':'identificador', 'sobrevive':'superviviente'})
df_nuevo.head(3)
```

	identificador	superviviente	clase	nombre	sexo	edad	hermanos	padres/ hijos	ticket	tarifa	camarote	puerto
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S

En nuestros ejemplos las etiquetas de las filas son números, pero también se pueden modificar con rename indicando index en vez de columns.

Discretización

A veces es interesante e incluso necesario discretizar variables, esto es, a partir de una variable continua separar en “contenedores” asignando a cada instancia 1 valor categórico. Imagina que quieres convertir la edad, una variable continua, en diferentes valores categóricos (niño, adocelescente, adulto, etc.).

Esta tarea se puede llevar a cabo creando una nueva columna, una función y el uso de map.

```python
# Generamos la función que recibe 1 valor y retorna 1 string
def convertir_edad(edad):
    if edad <= 12:
        return "niño"
    elif edad <= 18:
        return "adolescente"
    elif edad <=25:
        return "joven"
    elif edad <= 65:
        return "adulto"
    else:
        return "anciano"

# Usamos la función a través del map a una Series.
# map Devolverá una Series con los nuevos valores que asignaremos a una nueva columna
data_copy['EdadCat'] = data_copy['Age'].map(convertir_edad)

data_copy
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	EdadCat
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	adulto
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S	adulto

Discretización

Pandas tiene 2 métodos para hacer estas agrupaciones de forma automática y con mayor potencial: **cut** y **qcut**.

cut recibe una lista de valores (o una serie) y una lista con los límites para hacer los grupos. Devuelve un objeto "category" indicando el rango al que pertenece cada uno. Si un valor no está dentro de ningún grupo su resultado será NaN. Los agrupamientos se identifican por intervalos, de un lado abierto y del otro cerrado.

En matemáticas, un intervalo **abierto** (a, b), sus elementos son los números comprendidos entre a y b, sin incluir los propios números a y b. Un Intervalo **cerrado** [a, b], sus elementos son los números comprendidos entre a, b incluyendo a los propios a y b.

cut genera por defecto un intervalo abierto en la izquierda y cerrado en la derecha. Si deseamos que sea al contrario, incluimos el parámetro "right=False".

Por ejemplo, deseamos hacer grupos de [0-15), [15-25), [25,65), [65,130). La lista de valores límite son los que definen los grupos en orden, sin repetir número.

```
limites = [0, 15, 25, 65, 130]
pd.cut(df['edad'], limites, right=False)
```

Discretización

cut devuelve un objeto “category” indicando a qué categoría pertenece cada valor de la serie.

```
limites = [0, 15, 25, 65, 130]
pd.cut(df['edad'], limites, right=False)
```

```
0       [0.0, 15.0)
1      [25.0, 65.0)
2      [25.0, 65.0)
3      [25.0, 65.0)
4      [25.0, 65.0)
           ...
886    [25.0, 65.0)
887    [15.0, 25.0)
888             NaN
889    [25.0, 65.0)
890    [25.0, 65.0)
Name: edad, Length: 891, dtype: category
Categories (4, interval[int64, left]): [[0, 15) < [15, 25) < [25, 65) < [65, 130)]
```

Discretización

cut devuelve un objeto “category” indicando a qué categoría pertenece cada valor de la serie.

Este objeto se puede pasar al método de pandas pd.value_counts y nos devolverá cuántos elementos hay de cada categoría. El propio objeto también tiene un método value_counts().

```
limites = [0, 15, 25, 65, 130]
categorias = pd.cut(df['edad'], limites, right=False)

pd.value_counts(categorias)
```

```
edad
[25, 65)     425
[15, 25)     199
[0, 15)       79
[65, 130)     11
Name: count, dtype: int64
```

Discretización

Las etiquetas por defecto son los propios intervalos, pero se puede pasar una lista de strings.

Además, este objeto category se puede usar para crear una nueva columna de tipo categórica. La columna aún sigue siendo un objeto “category” en vez de un objeto Series.

```
limites = [0, 15, 25, 65, 130]
nombre_grupos = ['niño', 'joven', 'adulto', 'anciano']
categorias = pd.cut(df['edad'], limites, right=False, labels=nombre_grupos)
df['edad_cat'] = categorias
df.head(3)
```

	id	sobrevive	clase	nombre	sexo	edad	hermanos	padres/ hijos	ticket	tarifa	camarote	puerto	edad_cat
0	1	0	3	Braund, Mr. Owen Harris	male	1.0	1	0	A/5 21171	7.2500	NaN	S	niño
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	adulto
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	adulto

Discretización

Hasta ahora hemos indicado los bordes explícitamente para hacer las categorías. En lugar de ello podemos indicar cuántas categorías queremos y pandas se encargará de calcular la longitud de los contenedores en base a los valores mínimo y máximo de la columna.

En este ejemplo indicamos que queremos 4 grupos. El argumento nombrado para la cantidad de grupos es **bins**

```
pd.cut(df['edad'], 4, right=False)
```

```
0       [0.42, 20.315)
1      [20.315, 40.21)
2      [20.315, 40.21)
3      [20.315, 40.21)
4      [20.315, 40.21)
            ...
886    [20.315, 40.21)
887     [0.42, 20.315)
888                NaN
889    [20.315, 40.21)
890    [20.315, 40.21)
Name: edad, Length: 891, dtype: category
Categories (4, interval[float64, left]): [[0.42, 20.315) < [20.315, 40.21) < [40.21, 60.105) < [60.105, 80.08)]
```

Se puede indicar cuántos decimales de precisión deseamos con el argumento “precision”

Discretización

El método **cut** crea grupos no uniformes, esto es, puede que un grupo tenga el 60% de valores, otro grupo el 30%, y el 10% restante se lo repartan en otros 5 grupos.

Para realizar grupos más homogéneos en cuanto a cantidad de valores podemos utilizar **qcut**.

4 grupos con cut

```
categorias = pd.cut(df['edad'], 4)
categorias.value_counts()
```

```
edad
(20.315, 40.21]    384
(0.34, 20.315]     180
(40.21, 60.105]    128
(60.105, 80.0]      22
Name: count, dtype: int64
```

4 grupos con qcut

```
categorias = pd.qcut(df['edad'], 4)
categorias.value_counts()
```

```
edad
(20.0, 28.0]     182
(0.419, 20.0]    180
(38.0, 80.0]     177
(28.0, 38.0]     175
Name: count, dtype: int64
```

Valores atípicos

En estadística, un **valor atípico** (en inglés *outlier*) es una observación que es numéricamente distante del resto de los datos. Las estadísticas (media, varianza, etc.) derivadas de los conjuntos de datos que incluyen valores atípicos serán frecuentemente engañosas.

Por ejemplo, en el cálculo de la temperatura media de 10 objetos en una habitación, si la mayoría tienen entre 20 y 25 °C, pero hay un horno a 350 °C, la mediana de los datos puede ser 23, pero la temperatura media será 55. En este caso, la mediana refleja mejor la temperatura de la muestra al azar de un objeto que la media.

Los valores atípicos pueden ser indicativos de datos que pertenecen a una población diferente del resto de las muestras establecidas. O si se trata de valores medidos con sensores, pueden ser debidos a errores.

Los valores atípicos son en ocasiones una cuestión subjetiva. No existen formas automáticas para buscar y filtrar valores atípicos, sino que es una tarea manual (indexación booleana, consultas con query, usos de any() u all(), etc.)

Variables dummy o indicadoras

A veces es necesario convertir variables categóricas en una matriz dummy o indicadora.

Por ejemplo: Imagina que una variable categórica mide el precio y puede tener 3 valores (alto, medio, bajo). Una matriz indicadora a raíz de esa variable es una matriz de 3 columnas (alto, medio, bajo). Cada fila tendrá como valor un 1/True según corresponda a su precio, siendo 0/False en el resto.

	precio
0	alto
1	alto
2	medio
3	alto
4	bajo
5	medio

```
pd.get_dummies(df['precio'], prefix='precio')
```

	precio_alto	precio_bajo	precio_medio
0	True	False	False
1	True	False	False
2	False	False	True
3	True	False	False
4	False	True	False
5	False	False	True

Variables dummy o indicadoras

A veces una columna de tipo String puede indicar varias categorías. Por ejemplo, imagina la columna de tipo String "género" en un dataframe de películas. En este caso hay que usar un método diferente a través de los métodos str de pandas que manejan cadenas de caracteres, en concreto ***Series.str.get_dummies('separador')***

```
df = pd.read_csv('netflix_titles.csv')

df.head(5)
```

	show_id	type	title	director	cast	country	date_added	release_year	rating	duration	listed_in	description
0	s1	Movie	Dick Johnson Is Dead	Kirsten Johnson	NaN	United States	September 25, 2021	2020	PG-13	90 min	Documentaries	As her father nears the end of his life, filmm...
1	s2	TV Show	Blood & Water	NaN	Ama Qamata, Khosi Ngema, Gail Mabalane, Thaban...	South Africa	September 24, 2021	2021	TV-MA	2 Seasons	International TV Shows, TV Dramas, TV Mysteries	After crossing paths at a party, a Cape Town t...

```
dummies = df['listed_in'].str.get_dummies(',')

dummies.head(10)
```

	Anime Features	Children & Family Movies	Classic & Cult TV	Classic Movies	Comedies	Crime TV Shows	Cult Movies	Documentaries	Docuseries	Dramas	...	Sports Movies	Stand-Up Comedy	Stand-Up Comedy & Talk Shows	TV Action & Adventure	TV Comedies	I
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	

Variables dummy o indicadoras

Se puede combinar la creación de variables dummies con la función de discretización cut. Por ejemplo:

matriz_dummy = pd.get_dummies(pd.cut(df[‘columa’], bins/límites))

Una vez creada la matriz de variables dummies se pueden añadir al dataset original con el método join. En el momento se puede añadir un prefijo al nombre de las columnas.

df.join(matriz_dummy.add_prefix(“prefijo”))

La unión de dataframes se tratará en detalle más adelante.

Combinación y fusión de datasets

Pandas ofrece diversas herramientas para combinar datasets de forma eficiente, cada una con sus ventajas y aplicaciones específicas. A continuación, se presenta un resumen de los métodos más comunes:

1. `merge`**:** Une dataframes en función de una o varias columnas al estilo SQL. Permite realizar uniones de tipo "inner", "left", "right" o "outer", controlando qué filas se incluyen en el resultado. Si no se incluyen las columnas en parámetro “on”, pandas usa los nombres comunes en ambos datasets.

Sintaxis: pd.merge(df_left, df_right, on=column_name, how='inner')

df_left.merge(df_right, on=colum_name, how=’inner’)

Ejemplo: Dadas dos dataframes, clientes y pedidos.

clientes.csv:

ID	Nombre	Ciudad
1	Juan Pérez	Madrid
2	Ana López	Barcelona
3	Carlos García	Sevilla

pedidos.csv:

ID_pedido	ID_cliente	Producto	Precio
101	1	Camiseta	25
102	2	Zapatos	50
103	3	Gorra	15

Combinación y fusión de datasets

Para combinar estos datasets basándonos en la columna compartida `ID` mediante `merge`:

```
df_clientes = pd.read_csv('clientes.csv')
df_pedidos = pd.read_csv('pedidos.csv')

df_combinado = df_clientes.merge(df_pedidos, on='ID', how='inner')
```

ID	Nombre	Ciudad	ID_pedido	Producto	Precio
1	Juan Pérez	Madrid	101	Camiseta	25
2	Ana López	Barcelona	102	Zapatos	50
3	Carlos García	Sevilla	103	Gorra	15

Combinación y fusión de datasets

Tipos de merge:

1. Unión interna (inner): Tipo por defecto. Solo incluye las filas que tienen coincidencias en todas las columnas de unión especificadas. Si una fila no tiene coincidencia en la otra tabla, se descarta.

2. Unión izquierda (left): Incluye todas las filas de la tabla izquierda, independientemente de si tienen coincidencias en la tabla derecha. Las filas sin coincidencia en la derecha tendrán valores `NaN` en las columnas de la tabla derecha.

3. Unión derecha (right): Incluye todas las filas de la tabla derecha, independientemente de si tienen coincidencias en la tabla izquierda. Las filas sin coincidencia en la izquierda tendrán valores `NaN` en las columnas de la tabla izquierda.

4. Unión externa completa (outer): Incluye todas las filas de ambas tablas, sin importar si hay coincidencias o no. Las filas sin coincidencia en una tabla tendrán valores `NaN` en las columnas de la otra tabla.

Combinación y fusión de datasets

2. `concat`: Este método se utiliza para apilar datasets uno encima de otro si concatenamos por filas, o un dataset al lado de otro si concatenamos columnas. Es útil cuando los datasets no comparten columnas comunes.

Sintaxis: pd.concat([df1, df2], axis=0) # Concatenar filas (valor por defecto si no indicamos axis)

pd.concat([df1, df2], axis=1) # Concatenar columnas

Supongamos que queremos concatenar filas de dos datasets

df1:

Columna1	Columna2
A	10
B	20
C	30

df2:

Columna3	Columna4
D	40
E	50
F	60

Combinación y fusión de datasets

Para concatenar las filas de estos datasets verticalmente, podemos utilizar `concat`:

```
df_combinado = pd.concat([df1, df2])
```

En este ejemplo concatenamos filas apilando ambos datasets verticalmente. Los valores de columnas que no existen en la otra parte se rellenan con nulos.

Si existen columnas comunes, en esa columna no existirán nulos.

Al concatenar columnas, la unión se hace “casando” los índices de las filas.

Salida:

Columna1	Columna2	Columna3	Columna4
A	10	NaN	NaN
B	20	NaN	NaN
C	30	NaN	NaN
D	NaN	40	50
E	NaN	50	60
F	NaN	60	NaN

Combinación y fusión de datasets

3. **`combine_first`:** Este método combina datasets de forma similar a `concat`, pero prioriza los valores de la primera tabla en caso de existir columnas con el mismo nombre. Combina dataset coincidiendo por el índice de filas.

Sintaxis: DataFrame.combine_first(DataFrame)

4. **`join`:** Este método es un alias para `merge`, ofreciendo la misma funcionalidad con una sintaxis ligeramente diferente.

Sintaxis: df_left.join(df_right, on=column_name, how='inner')

Combinación y fusión de datasets

Elegir el método adecuado:

La elección del método adecuado depende de las características de los datasets y el resultado deseado. En general, se recomienda:

- **`merge`:** Para uniones precisas basándose en columnas comunes, con control sobre qué filas incluir.
- **`concat`:** Para apilar datasets sin necesidad de columnas comunes, concatenando filas o columnas.
- **`combine_first`:** Cuando se prioriza la información de un dataset sobre otro en caso de columnas con el mismo nombre.
- **`join`:** Como alternativa a `merge` con una sintaxis más concisa.

Agregación y funciones de grupos

`groupby:` es una función que permite agrupar y analizar datos de forma eficiente. Su poder reside en la capacidad de organizar un conjunto de datos en subgrupos basados en una o más columnas, y luego aplicar funciones de agregación o transformación a cada subgrupo.

2 Conceptos clave:

1. Hacer grupos en función del valor de una columna. Todas las filas que tienen el mismo valor en esa columna forman un grupo.
2. Función de agregación. Función que es aplicada en grupos. Las más habituales son mean(), sum(), count(), std(), max(), min() o funciones personalizadas.

Sintaxis:

```
df.groupby(column_name).agg(function_name)
```

Agregación y funciones de grupos

`Ejemplo:` Supongamos que tenemos un DataFrame `df` que contiene información sobre ventas de productos, con columnas como ID_producto, tipo, `precio` y `unidades_vendidas`.

ID_producto	Tipo	precio	unidades_vendidas
5	camiseta	20	4
3	camiseta	25	8
2	pantalón	30	4

Queremos calcular la cantidad total vendida de cada tipo. Debemos “agrupar” por tipo y calcular la función de agregación sobre la columna unidades_vendidas.

```
ventas_totales = df.groupby('producto')['unidades_vendidas'].sum()
```

Agregación y funciones de grupos

`Ejemplo 2:` Imagina que además tenemos la columna “provincia”. Queremos saber la cantidad de unidades_vendidas por tipo de producto y provincia.

Para ello debemos agrupar por dos columnas, tipo y provincia:

```
ventas_por_region = df.groupby(['producto', 'region'])['unidades_vendidas'].sum()
```

O mediante el método agg(funcion). Se pueden usar varias funciones de agregación, una por columna o varias a la misma columna.

Ejemplo:

```
ventas_por_producto_region = df.groupby(['producto', 'region'])[['unidades_vendidas', 'precio']].agg([sum, mean])
```

Aquí hacemos la suma de la columna ‘unidades_vendidas’ y mean sobre columna ‘precio’.

Funciones de cadena de caracteres en Pandas

Además de las funciones de la librería de Python para tratar cadenas de caracteres, podemos utilizar el conjunto de herramientas que trae Pandas. `Series.str` proporciona funciones para manipular cadenas de texto almacenadas en Series. Estas funciones permiten realizar diversas operaciones sobre los valores de texto, como:

1. Buscar y extraer patrones:

- `contains(pattern)`: Verifica si la cadena contiene un patrón específico.
- `startswith(pattern)`: Verifica si la cadena comienza con un patrón específico.
- `endswith(pattern)`: Verifica si la cadena termina con un patrón específico.
- `extract(pattern, expand=False)`: Extrae una parte coincidente de la cadena según un patrón regular.
- `replace(old, new, regex=False)`: Remplaza todas las ocurrencias de una cadena por otra.

Funciones de cadena de caracteres en Pandas

2. Convertir entre mayúsculas y minúsculas:

- `upper()`: Convierte toda la cadena a mayúsculas.
- `lower()`: Convierte toda la cadena a minúsculas.
- `title()`: Convierte la primera letra de cada palabra a mayúscula.

3. Eliminar espacios en blanco:

- `strip()`: Elimina espacios en blanco al principio y al final de la cadena.
- `lstrip()`: Elimina espacios en blanco al principio de la cadena.
- `rstrip()`: Elimina espacios en blanco al final de la cadena.

4. Buscar y contar subcadenas:

- `find(pattern)`: Devuelve la posición del primer carácter coincidente con el patrón.
- `rfind(pattern)`: Devuelve la posición del último carácter coincidente con el patrón.
- `count(pattern)`: Cuenta el número de ocurrencias de un patrón dentro de la cadena.

Funciones de cadena de caracteres en Pandas

5. Alinear texto:

- `pad(width, fillchar=' ')`: Rellena la cadena a la izquierda o derecha con un carácter de relleno hasta alcanzar el ancho especificado.
- `center(width, fillchar=' ')`: Centra la cadena dentro del ancho especificado, rellenando con un carácter de relleno.

6. Trabajar con caracteres:

- `len()`: Devuelve la longitud de la cadena.
- `isspace()`: Verifica si la cadena está compuesta solo por espacios en blanco.
- `isalnum()`: Verifica si la cadena está compuesta solo por letras y números.
- `isalpha()`: Verifica si la cadena está compuesta solo por letras.
- `isdigit()`: Verifica si la cadena está compuesta solo por números.

7. Convertir a otros tipos de datos:

- `astype(dtype)`: Convierte la cadena a un tipo de datos específico (por ejemplo, entero, flotante, fecha)

Funciones de cadena de caracteres en Pandas

Ejemplos: Dada la siguiente serie

```
nombres = pd.Series(['Ana López', 'Juan Pérez', 'Carlos García', 'María Gómez'])
```

- Eliminar espacios en blanco al final y principio de cada nombre:

  nombres.str.strip()

- Encontrar el apellido de cada persona:

  nombres.str.split(' ').str[-1]

- Contar la cantidad de nombres que comienzan con la letra "A":

  nombres.str.startswith('A')