AWAKELAB

BASECAMP

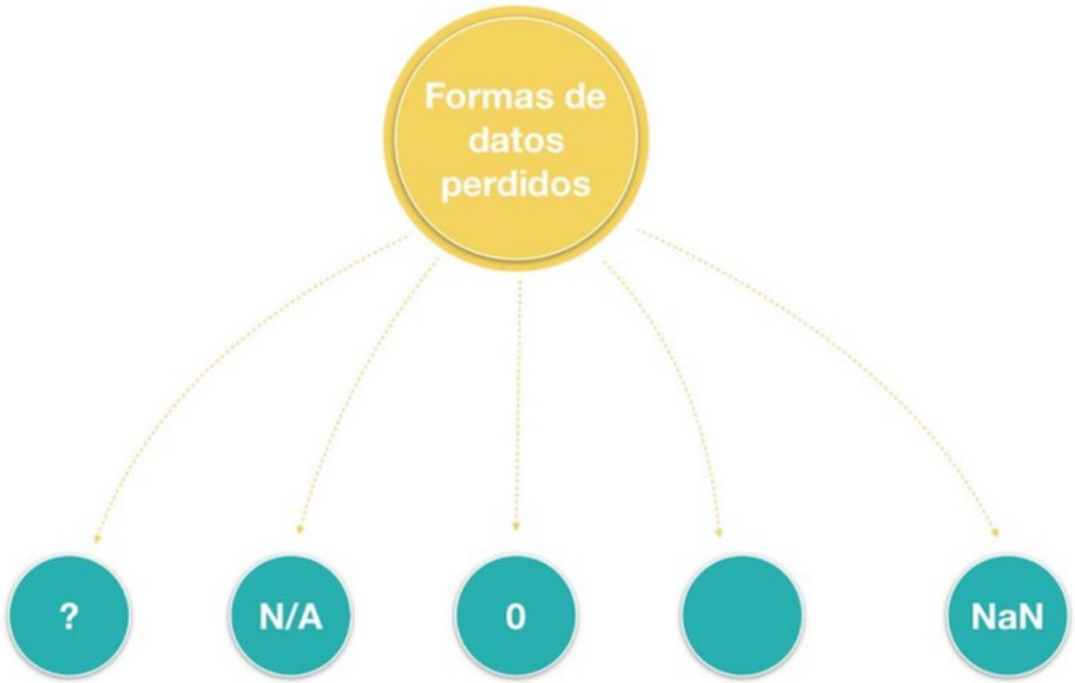
Ciencia de Datos

Obtención y Preparación de Datos

Objetivo de la jornada

- Aplica técnicas de limpieza y preparación de datos utilizando librerías Python para su depuración.

Manipulación de valores perdidos



En la gran mayoría de datos con los que trabajamos es muy probable que nos encontremos con valores perdidos, esto es algo muy normal. El valor faltante puede aparecer de distintas formas por ejemplo como un signo de interrogación, o N/A, como un 0 o simplemente como una celda en blanco, pero en su mayoría nos lo encontramos representado como NaN

que se refiere a “no un número”, pero ¿qué podemos hacer en estos casos?

Existen muchas maneras para corregir esto y todo dependerá de la persona que esté analizando y obviamente del problema en sí para seleccionar cual es la mejor opción para aplicar. Las opciones típicas que debes considerar son las siguientes:

- Lo primero que debes hacer es verificar con la persona o grupo que recopiló los datos si es posible que se pueda encontrar el valor real que está faltante.
- Otra posibilidad es simplemente eliminar los datos donde se encuentra ese valor perdido. Acá tienes dos opciones, la primera sería eliminar la variable faltante o eliminar la fila completa en donde está se encuentre. Esta opción es conveniente en los casos que no se tiene muchos datos faltantes. Toma en cuenta que se debe buscar hacer el menor impacto posible, por lo que se debe tener cuidado al momento de eliminar datos.
- Otra opción es reemplazar los datos, esto se puede hacer calculando el valor promedio de la variable completa, esta es una mejor opción ya que no se desperdician datos, sin embargo, es menos preciso ya que necesitamos reemplazar los datos faltantes con una conjetura que puede ser cierta como puede ser falsa. De igual forma, no siempre se puede calcular el valor del promedio ya que no siempre tenemos datos numéricos, pueden haber datos categóricos perdidos, para estos casos se sustituye el valor faltante por el más común de esa categoría, es decir el que se repita más veces.
- Y la última opción que tenemos es simplemente dejar los datos faltantes, así tal cual, como datos perdidos. En ocasiones esta puede ser una buena decisión, por eso se deben analizar cada uno de los casos de manera individual.


Corregir datos perdidos
Verificar con la persona que recopiló los datos si se pueda encontrar el valor faltante
Eliminar los datos donde se encuentra el valor perdido
Reemplazar los datos, esto se puede hacer calculando el valor promedio de la variable completa
Dejar los datos faltantes, así como datos perdidos

Ejemplo:

Vamos a crear un marco de datos con valores perdidos.

```
import pandas as pd
import numpy as np

df = pd.DatFrame({'column_a':[1, 2, 4, 4, np.nan,
np.nan, 6], 'column_b':[1.2, 1.4, np.nan, 6.2, None,
1.1, 4.3], 'column_c':['a', '?', 'c', 'd', '--',
np.nan, 'd'], 'column_d': [True, True, np.nan, None,
False, True, False]})

df
```

	column_a	column_b	column_c	column_d
0	1.0	1.2	a	True
1	2.0	1.4	?	True
2	4.0	NaN	c	NaN
3	4.0	6.2	d	None
4	NaN	NaN	--	False
5	NaN	1.1	NaN	True
6	6.0	4.3	d	False

np.nan, None y NaT (para los tipos datetime64 [ns]) son valores estándar perdidos para Pandas.

Nota: Un nuevo tipo de datos perdidos (<NA>) introducido con Pandas 1.0 que es una representación de valor perdido de tipo entero

np.nan es flotante, por lo que si los usa en una columna de enteros, se convertirán en un tipo de datos de punto flotante como puede ver en la "columna_a" del marco de datos que creamos. Sin embargo, <NA> se puede utilizar con números enteros sin provocar **upcasting**. Agreguemos

una columna más al marco de datos usando <NA> que se puede usar solicitando explícitamente el **dtype** Int64Dtype ().

```
new = pd.Series([1, 2, np.nan, 4, np.nan, 5], dtype =
pd.Int64Dtype())
df['column_e'] = new

df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	?	True	2
2	4.0	NaN	c	NaN	NaN
3	4.0	6.2	d	None	4
4	NaN	NaN	--	False	NaN
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	NaN

Encontrar valores perdidos.

Pandas proporciona funciones isnull () , isna () para detectar valores perdidos. Ambos hacen lo mismo.

df.isna () devuelve el marco de datos con valores booleanos que indican valores faltantes.

```
df.isna()
```

	column_a	column_b	column_c	column_d	column_e
0	False	False	False	False	False
1	False	False	False	False	False
2	False	True	False	True	True
3	False	False	False	True	False
4	True	True	False	False	True
5	True	False	True	False	False
6	False	False	False	False	True

También puede optar por utilizar notna() que es todo lo contrario de isna().

df.isna(). any() devuelve un valor booleano para cada columna. Si falta al menos un valor en esa columna, el resultado es Verdadero.

df.isna(). sum() devuelve el número de valores perdidos en cada columna.

```
df.isna().any()
```

```
column_a    True
column_b    True
column_c    True
column_d    True
column_e    True
dtype: bool
```

```
df.isna().any()
```

```
column_a    True
column_b    True
column_c    True
column_d    True
column_e    True
dtype: bool
```

Manejo de valores perdidos

No todos los valores faltantes vienen en formato np.nan o None agradable y limpio. Por ejemplo, "?" y los caracteres “- -” en la columna de nuestro marco de datos no nos brindan ninguna información o conocimiento valioso, por lo que esencialmente son valores perdidos. Sin embargo, estos caracteres no pueden ser detectados por Pandas como valor perdido.

Si sabemos qué tipo de caracteres se usaron como valores perdidos en el conjunto de datos, podemos manejarlos mientras creamos el marco de datos usando el parámetro na_values :

```
missing_values = ["?", "--"]

df_test = pd.read_csv("dataset.csv", na_values =
missing_values)
```

Otra opción es usar la función pandas **replace ()** para manejar estos valores después de que se crea un marco de datos

```
df.replace[{"?": np.nan, "--": np.nan}, inplace =
True]

df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	NaN	True	2
2	4.0	NaN	c	NaN	NaN
3	4.0	6.2	d	None	4
4	NaN	NaN	NaN	False	NaN
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	NaN

Hemos reemplazado las celdas no informativas con valores de NaN. El parámetro **inplace** guarda los cambios en el marco de datos. El valor predeterminado para inplace es False, por lo que si se establece en True, los cambios no se guardarán.

No existe una forma óptima de manejar los valores perdidos. Dependiendo de las características del conjunto de datos y la tarea, podemos optar por:

- Eliminar los valores perdidos
- Reemplazar valores perdidos

Eliminar los valores perdidos

	column_a	column_b	column_c	column_d
0	1.0	1.2	a	True
1	2.0	1.4	NaN	True
2	4.0	NaN	c	NaN
3	4.0	6.2	d	None
4	NaN	NaN	NaN	False
5	NaN	1.1	NaN	True
6	6.0	4.3	d	False

Podemos eliminar una fila o columna con valores perdidos usando la función **dropna** () . Cómo se usa el parámetro para establecer la condición para eliminar.

- how = 'any': eliminar si falta algún valor
- how = 'all': eliminar si faltan todos los valores

```
df.dropna(axis = 0, how = 'all', inplace = True)

df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	NaN	True	2
2	4.0	NaN	c	NaN	NaN
3	4.0	6.2	d	None	4
4	NaN	NaN	NaN	False	NaN
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	NaN

El parámetro de eje se utiliza para seleccionar la fila (0) o la columna (1).

Nuestro marco de datos no tiene una fila llena de valores faltantes, por lo que establecer cómo = 'todos' no eliminó ninguna fila. El valor predeterminado es 'cualquiera', por lo que no necesitamos especificar si queremos usar how = 'any':

```
df.dropna(axis = 0, inplace = True)

df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1

Se han eliminado las filas con al menos un valor faltante.

```
df.dropna(axis = 0, thresh = 3, inplace = True)
```

```
df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	NaN	True	2
3	4.0	6.2	d	None	4
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	NaN

Establecer el parámetro de umbral en 3 filas eliminadas con al menos 3 valores perdidos.

Los datos son un activo valioso por lo que no debemos renunciar a ellos fácilmente. Además, los modelos de aprendizaje automático casi siempre tienden a funcionar mejor con más datos. Por lo tanto, dependiendo de la situación, es posible que prefiramos reemplazar los valores perdidos en lugar de eliminarlos.

Reemplazo de valores perdidos

La función fillna() de Pandas maneja convenientemente los valores perdidos. Usando fillna(), los valores perdidos pueden ser reemplazados por un valor especial o un valor agregado como media, mediana. Además, los valores perdidos se pueden reemplazar con el valor anterior o posterior, lo cual es bastante útil para conjuntos de datos de series de tiempo.

- Reemplace los valores perdidos con un escalar:

```
df.fillna(25)
```

	column_a	column_b	column_c	column_d
0	1.0	1.2	a	True
1	2.0	1.4	25	True
2	4.0	25.0	c	25
3	4.0	6.2	d	25
4	25.0	25.0	25	False
5	25.0	1.1	25	True
6	6.0	4.3	d	False

- fillna () también se puede usar en una columna en particular:

```
mean = df['column_a'].mean()
df['column_a'].fillna(mean)
```

```
0    1.0
1    2.0
2    4.0
3    4.0
4    3.4
5    3.4
6    6.0
Name: column_a, dtype: float64
```

- Usando el parámetro de método , los valores perdidos se pueden reemplazar con los valores antes o después de ellos.

```
df.fillna(axis = 0, method = 'ffill')
```

	column_a	column_b	column_c	column_d
0	1.0	1.2	a	True
1	2.0	1.4	a	True
2	4.0	1.4	c	True
3	4.0	6.2	d	True
4	4.0	6.2	d	False
5	4.0	1.1	d	True
6	6.0	4.3	d	False

Si hay muchos valores perdidos consecutivos en una columna o fila, es posible que desee **limitar** el número de valores perdidos para que se llenen hacia adelante o hacia atrás.

```
df.fillna(axis = 0, method = 'ffill', limit = 1)
```

	column_a	column_b	column_c	column_d
0	1.0	1.2	a	True
1	2.0	1.4	a	True
2	4.0	1.4	c	True
3	4.0	6.2	d	None
4	4.0	6.2	d	False
5	NaN	1.1	NaN	True
6	6.0	4.3	d	False

El parámetro de límite se establece en 1, por lo que solo se completa hacia adelante un valor faltante.

Imputación de datos

La imputación de datos es un método en el que los valores faltantes en cualquier variable o marco de datos (en el aprendizaje automático) se rellenan con algunos valores numéricos para realizar la tarea. Con este método, el tamaño de la muestra sigue siendo el mismo, solo los **espacios en blanco** que *faltaban ahora se llenan con algunos valores*. *Este método es fácil de usar pero reduce la varianza del conjunto de datos.*

Puede haber varias razones para imputar datos, muchos conjuntos de datos del mundo real (sin hablar de CIFAR o MNIST) que contienen valores faltantes que pueden estar en cualquier forma, como espacios en blanco, NaN, ceros, cualquier número entero o cualquier símbolo categórico. En lugar de simplemente eliminar las Filas o Columnas que contienen los valores perdidos, lo que conlleva la pérdida de datos que pueden ser valiosos, una mejor estrategia es imputar los valores perdidos.

Tener un buen conocimiento teórico es increíble, pero implementarlos en código en un proyecto de aprendizaje automático en tiempo real es algo completamente diferente. Es posible que obtenga resultados diferentes e inesperados basados en diferentes problemas y conjuntos de datos. Entonces, como un bono, también estoy agregando los enlaces a los diversos cursos que me han ayudado mucho en mi viaje para aprender ciencia de datos y ML, experimentar y comparar diferentes estrategias de imputación de datos que me llevaron a escribir este artículo sobre comparaciones entre diferentes datos. Métodos de imputación.

Hay dos aproximaciones comúnmente usadas para la imputación de los valores perdidos.

La primera técnica consiste en rellenar estos valores con la media (o mediana) de los datos de la variable en el caso de que se trate de una variable numérica. Para el caso de las variables categóricas imputamos los valores perdidos con la moda de la variable.

Pandas nos ofrece una función (fillna) para realizar esta imputación de manera sencilla. A continuación se puede ver como imputamos la variable numérica bore con la media y la variable categórica num-of-doors con el valor más frecuente:

```
#Imputamos la variable bore con la media

df['bore'].fillna(df['bore'].mean, inplace=True)

print("valores perdidos en bore: " +
str(df['bore'].isnull().sum()))


#Imputamos la variable num-of-doors con la moda

df['num-of-doors'].fillna(df['num-of-
doors'].mode()[0], inplace=True)

print("valores perdidos en num-of-doors: " +
str(df['num-of-doors'].isnull().sum()))

valores perdidos en bore: 0

valores perdidos en num-of-doors: 0
```

Para este ejemplo práctico hemos calculado la media/moda de todos los datos. Sin embargo, en un ejemplo real y para evitar la fuga de datos (data leakage en inglés), es importante dividir los datos en datos de entrenamiento y datos de test, calcular la media/moda únicamente de los datos de entrenamiento y aplicarla a los datos de prueba.

Otra técnica más avanzada consiste en el uso de modelos predictivos para estimar los valores perdidos. Un modelo no paramétrico muy popular para estos casos es el k-nearest neighbors, donde se estima el valor perdido como la media (en el caso de las variables numéricas) de los valores de los k vecinos u observaciones más cercanos. Asimismo, para las

variables categóricas, se utiliza la clase mayoritaria de entre los k más cercanos.

La librería scikit-learn nos proporciona la clase KNNImputer para hacer uso de este modelo en la imputación de missing values. Esta clase usa por defecto la distancia euclidiana, pero podemos elegir la que prefiramos modificando el parámetro metric. Asimismo, también podemos elegir el número de vecinos con el argumento n_neighbors y en esta ocasión usaremos 5. Puedes ver en el siguiente código como utilizamos este método para imputar los valores perdidos en la variable normalized-losses:

```
from sklearn.impute import KNNImputer

# Construimos el modelo

imputer = KNNImputer(n_neighbors=5,
weights="uniform")



# Ajustamos el modelo e imputamos los missing values

imputer.fir(df[["normalized-losses"]])

df["normalized-losses"] =
imputer.transform(df[["normalized-losses"]]).ravel()

print("Valores perdidos en normalized-losses: " +
str(df['normalized-losses'].isnull().sum()))

Valores perdidos en normalized-losses: 0
```

Imputación de datos cualitativos

Si los datos nos dicen en cual de determinadas categorías no numéricas nuestros ítems van a caer, entonces estamos hablando de datos cualitativos o categóricos; ya que los mismos van a representar

determinada cualidad que los ítems poseen. Dentro de esta categoría vamos a encontrar datos como: el sexo de una persona, el estado civil, la ciudad natal, o los tipos de películas que le gustan. Los datos categóricos pueden tomar valores numéricos (por ejemplo, "1" para indicar "masculino" y "2" para indicar "femenino"), pero esos números no tienen un sentido matemático.

Para ejemplificar el análisis, vamos a utilizar nuestras habituales librerías científicas NumPy, Pandas, Matplotlib y Seaborn. También vamos a utilizar la librería pydataset, la cual nos facilita cargar los diferentes dataset para analizar.

La idea es realizar un análisis estadístico sobre los datos de los sobrevivientes a la tragedia del Titanic.

La tragedia del Titanic

El hundimiento del Titanic es uno de los naufragios más infames de la historia. El 15 de abril de 1912, durante su viaje inaugural, el Titanic se hundió después de chocar con un iceberg, matando a miles de personas. Esta tragedia sensacional conmocionó a la comunidad internacional y condujo a mejores normas de seguridad aplicables a los buques. Una de las razones por las que el naufragio dio lugar a semejante cantidad de muertes fue que no había suficientes botes salvavidas para los pasajeros y la tripulación. Aunque hubo algún elemento de suerte involucrado en sobrevivir al hundimiento, algunos grupos de personas tenían más probabilidades de sobrevivir que otros, como las mujeres, los niños y la clase alta.

El siguiente **dataset** proporciona información sobre el destino de los pasajeros en el viaje fatal del trasatlántico Titanic, que se resume de acuerdo con el nivel económico (clase), el sexo, la edad y la supervivencia.

```
import numpy as np
import pandas as pd
from pydataset import data

titanic = data('deep', desat = 0.6)
```

```
titanic.head(10)
```

Out[3]:

	class	age	sex	survived
1	1st class	adults	man	yes
2	1st class	adults	man	yes
3	1st class	adults	man	yes
4	1st class	adults	man	yes
5	1st class	adults	man	yes
6	1st class	adults	man	yes
7	1st class	adults	man	yes
8	1st class	adults	man	yes
9	1st class	adults	man	yes
10	1st class	adults	man	yes

El problema con datos como estos, y en general con la mayoría de las tablas de datos, es que nos presentan mucha información y no nos permiten ver que es lo que realmente sucede o sucedió. Por tanto, deberíamos procesar de alguna manera para hacernos una imagen de lo que los datos realmente representan y nos quieren decir; y qué mejor manera para hacernos una imagen de algo que utilizar visualizaciones. Una buena visualización de los datos puede revelar cosas que es probable que no podamos ver en una tabla de números y nos ayudará a pensar con claridad acerca de los patrones y relaciones que pueden estar escondidos en los datos. También nos va a ayudar a encontrar las características y patrones más importantes o los casos que son realmente excepcionales y no deberíamos de encontrar.

Tabla de Frecuencia

Para hacernos una imagen de los datos, lo primero que tenemos que hacer es agruparlos. Al armar diferentes grupos nos vamos acercando a la comprensión de los datos. La idea es ir amontonamos las cosas que parecen ir juntas, para poder ver cómo se distribuyen a través de las diferentes categorías. Para los **datos categóricos**, agrupar es fácil; simplemente debemos contar el número de ítems que corresponden a cada categoría y apilarlos. Una forma en la que podemos agrupar nuestro **dataset** del Titanic es contando las diferentes clases de pasajeros. Podemos organizar estos conteos en una tabla de **frecuencia**, que registra los totales y los nombres de las categorías utilizando la función **value_counts** que nos proporciona **Pandas** del siguiente modo:

```
pd.value_counts(titanic['class'])
```

```
Out[4]:  3rd class    706
         1st class    325
         2nd class    285
         dtype: int64
```

Contar la cantidad de apariciones de cada categoría puede ser útil, pero a veces puede resultar más útil saber la fracción o proporción de los datos de cada categoría, así que podríamos entonces dividir los recuentos por el total de casos para obtener los porcentajes que representa cada categoría.

Una tabla de frecuencia relativa muestra los porcentajes, en lugar de los recuentos de los valores en cada categoría. Ambos tipos de tablas muestran cómo los casos se distribuyen a través de las categorías. De esta manera, ellas describen la distribución de una variable categórica, ya que enumeran las posibles categorías y nos dicen con qué frecuencia se produce cada una de ellas.

```
100*titanic['class'].value_counts() / len(titanic['class'])
```

```
Out[5]:  3rd class    53.647416
         1st class    24.696049
         2nd class    21.656535
         dtype: float64
```

Técnicas avanzadas de imputación con k-NN

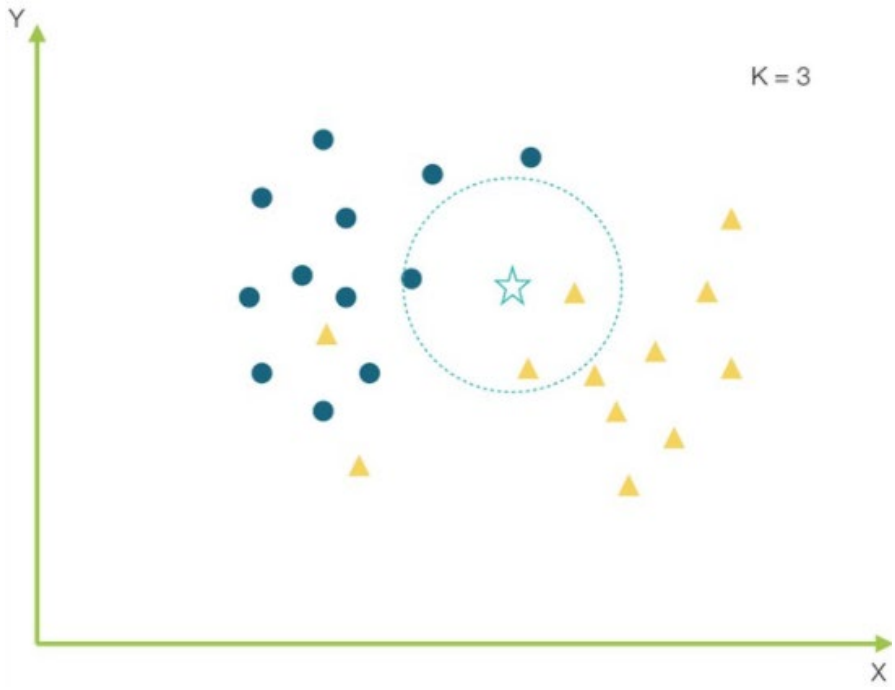
K vecinos más cercanos es uno de los algoritmos de clasificación más básicos y esenciales en **Machine Learning**. Pertenece al dominio del **aprendizaje supervisado** y encuentra una aplicación intensa en el reconocimiento de patrones, la minería de datos y la detección de intrusos.

El **clasificador KNN**, por sus siglas en inglés, es también un algoritmo de aprendizaje no paramétrico y basado en instancias:

- No paramétrico significa que no hace suposiciones explícitas sobre la forma funcional de los datos, evitando modelas mal la distribución subyacente de los datos. Por ejemplo, supongamos que nuestros datos son altamente no gaussianos, pero el modelo de Machine Learning que elegimos asume una forma gaussiana. En este caso, nuestro algoritmo haría predicciones extremadamente pobres.

- El aprendizaje basado en la instancia significa que nuestro algoritmo no aprende explícitamente un modelo. En lugar de ello, opta por memorizar las instancias de formación que posteriormente se utilizan como "conocimiento" para la fase de predicción. Concretamente, esto significa que solo cuando se realiza una consulta a nuestra base de datos, es decir, cuando le pedimos que predique una etiqueta con una entrada, el algoritmo utilizará las instancias de formación para dar una respuesta.

Cabe señalar que la fase de formación mínima de KNN se realiza tanto a un coste de memoria, ya que debemos almacenar un conjunto de datos potencialmente enorme, como un coste computacional durante el tiempo de prueba, ya que la clasificación de una observación determinada requiere un agotamiento de todo el conjunto de dato. En la práctica, esto no es deseable, ya que normalmente queremos respuestas rápidas.



Supongamos que Z es el punto el cual se necesita predecir. Primero, se encuentra el punto K más cercano a Z y luego se clasifican los puntos para el voto mayoritario de sus vecinos K. Cada objeto vota por su clase y la clase con más votos se toma como la predicción. Para encontrar los puntos similares más cercanos, se encuentra la distancia entre puntos utilizando medidas de distancias.

Una opción popular es la distancia euclidiana, pero también hay otras medidas que pueden ser más adecuadas para un entorno dado e incluyen la distancia de Mahattan y Minkowski.

Distancia Euclidiana

$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Distancia Manhattan

$$\sum_{i=1}^{k}|x_i - y_i|$$

Distancia Minkowski

$$[\sum_{i=1}^{k}(|x_i - y_i|)^4]^{\frac{1}{4}}$$

En resumen, KNN tiene los siguientes pasos básicos:

- Calcular la distancia
- Encontrar sus vecinos más cercanos
- Votar por las etiquetas

¿Cómo se decide el número de vecinos en KNN?

Ya que conocemos cómo funciona este algoritmo, es momento de saber cómo se define K. El número de vecinos (K) es un hiperparámetro que se debe elegir en el momento de la construcción del modelo. Puedes pensar en K como una variable de control para el modelo de predicción.

La investigación ha demostrado que no existe un número óptimo de vecinos que se adapte a todo tipo de conjuntos de datos. Cada conjunto de datos tiene sus propios requisitos. Se ha demostrado que una pequeña cantidad de vecinos son los más flexibles, que tendrán un bajo sesgo, pero una alta varianza, y un gran número de vecinos tendrán un límite de decisión más suave, lo que significa una varianza más baja pero un sesgo más alto.

Generalmente, se recomienda elegir un número impar si el número de clases es par. También puede comprobar generando el modelo en diferentes valores de K y comprobar su rendimiento.

Transformación de la data

La transformación de los datos es necesaria cuando entre las variables existen diferentes escalas o existen demasiadas o pocas variables, entonces se realiza una normalización o una estandarización de los datos mediante técnicas de reducción o aumento de la dimensión, así como el escalamiento simple o multidimensional.

Si en el análisis exploratorio se indica la necesidad de transformar algunas variables, se podrán aplicar algunas de estas cuatro transformaciones:

- Transformaciones lógicas
- Transformaciones lineales
- Transformaciones algebraicas
- Transformaciones no lineales

Estas fases mencionadas en los puntos anteriores, constituyen el proceso **de pre-procesamiento** de los datos y para aplicar estos conceptos, se presenta el siguiente ejemplo:

Ejemplo de Pre-procesamiento

Supongamos la siguiente tabla de datos, que representa información de una tienda que relaciona datos de clientes que compraron y clientes que no compraron:

No	País	Edad	Salario	Compra
1	Francia	44	72000	No
2	España	27	48000	Si
3	Alemania	30	54000	No
4	España	38	61000	No
5	Alemania	40		Si
6	Francia	35	58000	Si
7	España		52000	No
8	Francia	48	79000	Si
9	Alemania	50	83000	No
10	Francia	37	67000	Si

Como podemos observar, existen registros con datos faltantes. El cliente del registro o renglón 5 no tiene salario y el cliente del renglón 7 no tiene edad.

Cuando se tienen datos faltantes, hay varias estrategias que se pueden seguir, sobre todo si se ha comprobado la aleatoriedad de dichos datos ausentes. Una estrategia es usar el método de **aproximación de casos completos** que consiste en incluir en el análisis sólo los casos con datos completos, únicamente filas cuyos valores para todas las variables son válidos.

En este caso tenemos que validar si la muestra se ve afectada o no al eliminar los casos incompletos.

La alternativa a los métodos de eliminación de datos es la **imputación de la información faltante**, donde el objetivo es estimar los valores ausentes basados en valores válidos de otras variables o casos.

Un método para llevar a cabo la imputación de la información faltante es el **método de sustitución de casos**. Este consiste en sustituir los datos ausentes con datos de observaciones no muestrales, es decir, que no pertenecen a la muestra.

El **método de sustitución por la media o la mediana**, sustituye los datos ausentes por el valor de la media o la mediana de todos los valores válidos de su variable correspondiente. Cuando existen valores extremos o atípicos, los datos ausentes se sustituyen por la mediana, en caso contrario se utiliza la media.

El método de imputación por interpolación sustituye cada valor ausente por la media o la mediana de un cierto número de datos u observaciones adyacentes a él, sobre todo cuando hay demasiada variabilidad en los datos. Si no existe dicha variabilidad, entonces se sustituyen por el valor resultante de realizar una interpolación con los valores adyacentes.

Adicionalmente, también existe el **método de sustitución por un valor constante** y como su nombre lo indica, los datos ausentes se sustituyen por un valor constante, válido para la variable en cuestión, derivado de fuentes externas o de una investigación previa.

Por último, podemos utilizar también el **método de imputación por regresión** y éste utiliza el método de la regresión para calcular o estimar los valores ausentes basados en su relación con otras variables del conjunto de datos.

En nuestro ejemplo vamos a utilizar la media como valor para imputar los datos faltantes tanto de edad como de salario. En este sentido el promedio de valores de la edad es: 38.77 y para el salario, el valor promedio de los valores presentes en la muestra es: 63,777.77, por lo que la tabla ahora queda de la siguiente manera:

No	País	Edad	Salario	Compra
1	Francia	44	72000	No
2	España	27	48000	Si
3	Alemania	30	54000	No
4	España	38	61000	No
5	Alemania	40	**63777.77**	Si
6	Francia	35	58000	Si
7	España	**38.77**	52000	No
8	Francia	48	79000	Si
9	Alemania	50	83000	No
10	Francia	37	67000	Si

Ya tenemos la muestra completa y ahora continuamos con las **variables categóricas** (País y Compra).

Para la variable Compra, que tiene valores categóricos Si o No, podemos codificarla como 1 y 0 representando 1 para el valor Si y 0 para el valor No. Para el caso de la variable País, el caso es un poco distinto dado que si codificamos como 0, 1 y 2, el país con el valor 2 tendría más peso que el país con el valor 0 por lo que la estrategia para este tipo de variables es diferente.

Dentro de las transformaciones lógicas, las variables de intervalo se pueden convertir en ordinales como las variables Talla o en nominales como Color, y crear variables ficticias o dummy.

País	Francia	España	Alemania
Francia	1	0	0
España	0	1	0
Alemania	0	0	1

La variable País la cambiamos por tres variables (Dummy) que son los valores de la variable País. En lugar de utilizar la variable País, utilizamos ahora las 3 variables nuevas y la tabla quedaría de la siguiente forma:

No	Francia	España	Alemania	Edad	Salario	Compra
1	1	0	0	44	72000	0
2	0	1	0	27	48000	1
3	0	0	1	30	54000	0
4	0	1	0	38	61000	0
5	0	0	1	40	63777.77	1
6	1	0	0	35	58000	1
7	0	1	0	38.77	52000	0
8	1	0	0	48	79000	1
9	0	0	1	50	83000	0
10	1	0	0	37	67000	1

Dado que los datos de edad y salario mantienen una escala diferente y en las ecuaciones de la regresión o de algún otro método de clasificación y/o predicción, dada la distancia euclidiana entre dos puntos, el valor del salario podría hacer que la edad deje de ser representativa o importante para el análisis, lo más conveniente es hacer una transformación de escalas, ya sea una estandarización o una normalización de los datos.

Estandarización	Normalización
$x_{stand} = \frac{x - mean(x)}{Standard\ Dev(x)}$	$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$

Para este ejemplo usaremos la normalización para ajustar las escalas de todas las variables que está dado por el valor actual de la muestra menos el valor mínimo de todo el conjunto de datos de esa variable entre la diferencia del valor máximo y el valor mínimo.

Una vez aplicada la normalización, la tabla resultante es la siguiente:

Francia	Alemania	España	Age	Salary	Purchased
1	0	0	0.72003861	0.71101283	0
0	0	1	-1.62356783	-1.36437577	1
0	1	0	-1.20999022	-0.84552862	0
0	0	1	-0.1071166	-0.24020695	0
0	1	0	0.1686018	-6.0532E-07	1
1	0	0	-0.52069421	-0.49963052	1
0	0	1	-0.00096501	-1.01847767	0
1	0	0	1.27147542	1.3163345	1
0	1	0	1.54719383	1.6622326	0
1	0	0	-0.2449758	0.2786402	1

Cuando las variables categóricas generan muchas variables dummy podemos utilizar las técnicas de reducción de la dimensión para hace nuestro conjunto de datos más manejable.

Remover duplicados

Un problema con el que nos podemos encontrar de forma más frecuente es la identificación de valores únicos en una lista. Ya que la existencia de valores duplicados no es de interés para el análisis que se desea realizar. Por eso, saber cómo eliminar los valores duplicados en una lista es un truco que debemos tener en nuestro arsenal. Otro truco que también puede ser de interés es contar el número de ocurrencias de cada valor en una lista, lo que también se comentará en esta entrada.

Supongamos que tenemos una lista con diferentes números. Para obtener una nueva lista sin valores duplicados solamente tenemos que crear una nueva lista vacía recorriendo la lista inicial y añadiendo los valores a la nueva lista si el valor no se encuentra en esta. Lo que se puede resolver con el siguiente código.

```
data = [1, 3, 2, 4, 7, 3, 2, 2, 1, 4]

result = []
for item in data:
      if item not in result:
            result.append(item)

result

[1, 3, 2, 4, 7]
```

En donde se puede apreciar que la lista **result** contiene los valores de la lista inicial (data) sin valores duplicados. Para lo que solamente se ha utilizado código básico de Python.

Una alternativa a la construcción de un bucle para eliminar los valores duplicados de una lista en Python es la instrucción set. Lo que genera una colección desordenada de objetos diferentes.

```
set(data)

{1, 2, 3, 4, 7}
```

Si lo que necesitamos es una lista para trabajar con ella solamente necesitamos convertirla con **list**

```
list(set(data))

[1, 2, 3, 4, 7]
```

La principal diferencia respecto al caso anterior es que el resultado no respeta el orden de aparición en la lista original, sino que los valores se ordenarán según el valor. Por lo que este método puede no ser adecuado cuando es necesario conservar el orden de aparición de valores.

En la librería Pandas existe un método con el que obtener los valores únicos de una lista de valores. Un método que se llama unique.

```
import pandas as pd
pd.unique(data)
```

Utilización de funciones de mapeo.

map es una función incorporada en Python, que usa una función y un objeto iterable como parámetros. El formulario es el siguiente:

- map(aFunction,iterable)

Muestra 1

```
# 1. Primero declara una lista

a = [10,20,30]


# Ahora, llame a la función de mapa en la declaración de impresión

print(list(map(lambda x: x**2, a)))

[100,400,900]
```

Muestra 2

```
print(list(map(lambda x: x**2, a)))

print(list(map(lambda x: x**3, a)))

# Utilice la función de mapa para cambiar el segmento de código de suma 1,16 una sola línea

print(sum(list(map(lambda x: x**2, a))))

print(sum(list(map(lambda x: x**3, a))))


[100,400,900]

[1000, 8000, 27000]
```

```
1400

36000
```

Muestra 3

```
# Pasamos dos secuencias a la función de mapa. pow
calcula secuencialmente la potencia del exponente

# Las dos listas tienen el mismo tamaño. Si los
tamaños son inconsistentes, Python lo completará
automáticamente

a = [10,20,30]

b = [1,2,3]

print(list(map(pow,a,b)))


[10, 400, 27000]
```

Renombrando índices y ejes

Una columna de un marco de datos se puede cambiar usando la posición en la que se encuentra conocida como su índice. Con solo usar el índice, se puede cambiar el nombre de una columna.

El método pandas **rename()** se utiliza para cambiar el nombre de cualquier índice, columna o fila.

Sintaxis: renombrar (asignador = None, índice = None, columnas = None, eje = None, copiar = Verdadero, en lugar = False, nivel = None)

Parámetros:

- **asignador, índice y columnas:** valor del diccionario, la clave se refiere al nombre anterior y el valor se refiere al nuevo nombre. Solo se puede utilizar uno de estos parámetros a la vez.
- **eje:** int o valor de cadena, 0 / 'fila' para Filas y 1 / 'columnas' para Columnas
- **copiar:** copia los datos subyacentes si es verdadero.
- **inplace:** realiza cambios en el marco de datos original si es verdadero.
- **nivel:** se utiliza para especificar el nivel en caso de que el marco de datos tenga un índice de niveles múltiples.

Tipo de retorno: marco de datos con nuevos nombres

Ejemplo 1: Cambiar los nombres de ambas columnas entre sí utilizando el índice de columna.

```
import pandas as pd

df = pd.DataFrame({'a':[1, 2], 'b': [3, 4]})

df.columns.values[0] = 'b'
df.columns.values[1] = 'a'

display(df)
```

	b	a
0	1	3
1	2	4

Ejemplo 2: uso de otro método para cambiar el nombre de la columna con index.

```
import pandas as pd

df = pd.DataFrame({'a':[1, 2], 'b': [3, 4]})

su = df.rename(columns = {df.column[1]: 'new'})

display(su)
```

	a	new
0	1	3
1	2	4

Ejemplo 3: Cambiar el nombre de dos o más columnas en un solo comando utilizando un número de índice.

```
import pandas as pd

df = pd.DataFrame({'a':[1, 2], 'b': [3, 4], 'c': [7,
8]})

mapping = {df.columns[0]: 'new0', df.columns[1]:
'new1'}
su = df.rename(columns = mapping)

display(su)
```

	new0	new1	c
0	1	3	7
1	2	4	8

Ejemplo 4: Cambiar el nombre de la columna con un número de índice del archivo CSV.

data1.csv

	A	B	C
1	Name	Marks	Country
2	Mayank	455	Jaipur
3	Amit	555	Patna
4	Hari	645	Ooty
5	Suraj	255	Rachi

```
import pandas as pd

df1 = pd.read_csv("data1.csv")
df1.columns.values[2] = "city"

display(df1)
```

	Name	Marks	city
0	Mayank	455	Jaipur
1	Amit	555	Patna
2	Hari	645	Ooty
3	Suraj	255	Rachi

Discretización y Binning

Los datos del mundo real tienden a ser ruidosos. Los datos ruidosos son datos con una gran cantidad de información adicional sin sentido llamada ruido. Las rutinas de limpieza de datos intentan suavizar el ruido mientras identifica valores atípicos en los datos.

Hay tres técnicas de suavizado de datos de la siguiente manera:

1. **Binning:** Los métodos de binning suavizan un valor de datos ordenados consultando su «vecindad», es decir, los valores que lo rodean.
2. **Regresión:** ajusta los valores de los datos a una función. La regresión lineal implica encontrar la línea «mejor» para ajustar dos atributos(o variables) de modo que un atributo pueda usarse para predecir el otro.
3. **Análisis de valores atípicos:** los valores atípicos pueden detectarse mediante agrupación, por ejemplo, cuando los valores similares se organizan en grupos o «**agrupaciones**». De manera intuitiva, los valores que quedan fuera del conjunto de grupos pueden considerarse valores atípicos.

Método de agrupación para el suavizado de datos:

En este método los datos se clasifican primero y luego los valores ordenados se distribuyen en un número de *cubos* o *contenedores*. A medida que los métodos de agrupamiento consultan la vecindad de valores, realizan un suavizado local.

Básicamente, existen dos tipos de enfoques de agrupamiento:

1. Intervalos de igual ancho(o distancia): el enfoque de agrupamiento más simple es dividir el rango de la variable en k intervalos de igual ancho. El ancho del intervalo es simplemente el rango [A, B] de la variable dividido por k,

$$w = (BA) / k$$

Por lo tanto, el rango i- ésimo del intervalo será [A +(i-1)w, A + iw] donde i = 1, 2, 3… ..k

Los datos sesgados no pueden manejarse bien con este método.

2. Agrupación de igual profundidad(o frecuencia): En la agrupación de igual frecuencia dividimos el rango [A, B] de la variable en intervalos que contienen (aproximadamente) el mismo número de puntos; Puede que no sea posible obtener la misma frecuencia debido a valores repetidos.

¿Cómo realizar el suavizado de los datos?

Hay tres enfoques para realizar el suavizado:

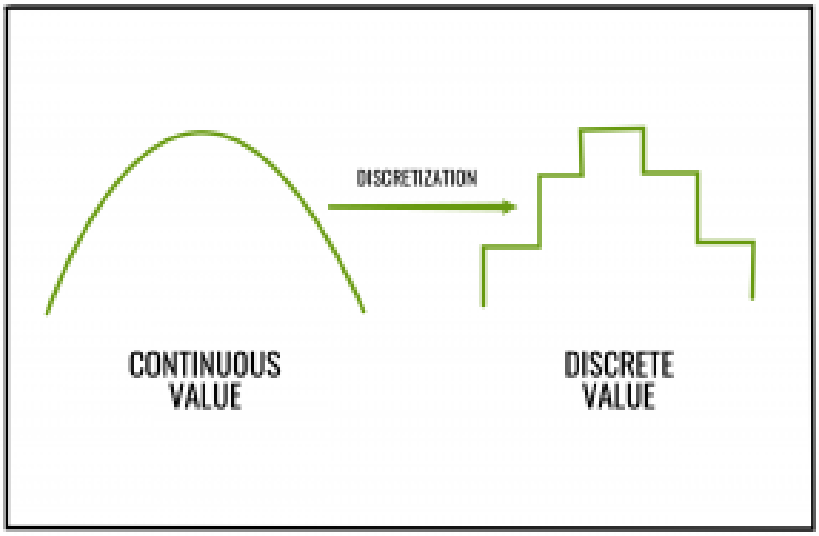
1. Suavizado por medio de contenedor: en el suavizado por medio de contenedor, cada valor de un contenedor se reemplaza por el valor medio del contenedor.
2. Suavizado por la mediana de intervalo: en este método, cada valor de intervalo se reemplaza por su valor de mediana de intervalo.
3. Suavizado por límite de contenedor: en el suavizado por límites de contenedor, los valores mínimo y máximo en un contenedor determinado se identifican como los límites de contenedor. Luego, cada valor de ubicación se reemplaza por el valor de límite más cercano.

Datos ordenados por precio (en dólares): 2, 6, 7, 9, 13, 20, 21, 25, 30

```
Partición utilizando el enfoque de igual frecuencia:
Bandeja 1: 2, 6, 7
Bin 2: 9, 13, 20
Papelera 3:21, 24, 30
Suavizado por bin significa:
Bandeja 1: 5, 5, 5
Papelera 2:14, 14, 14
Papelera 3:25, 25, 25
Suavizado por mediana de bandeja:
Bandeja 1: 6, 6, 6
Papelera 2:13, 13, 13
Papelera 3:24, 24, 24
Suavizado por límite de contenedor:
Bin 1: 2, 7, 7
Bin 2: 9, 9, 20
Papelera 3:21, 21, 30
```

El binning también se puede utilizar como técnica de discretización . Aquí, la discretización se refiere al proceso de convertir o dividir atributos, características o variables continuas en atributos / características / variables / intervalos discretizados o nominales.

Por ejemplo, los valores de los atributos se pueden discretizar aplicando *agrupaciones de igual ancho* o *igual frecuencia* y luego reemplazando cada valor de agrupación por la media o mediana de agrupación, como en el suavizado por medias de agrupación o el suavizado por medianas de agrupación, respectivamente. Luego, los valores continuos se pueden convertir a un valor nominal o discretizado que es el mismo que el valor de su bin correspondiente.



A continuación se muestra la implementación de Python:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn import linear_model
import statistics
import math
from collections import OrderedDict


x =[]
print("Entrega en número de datos")
x = list(map(float, input().split()))


print("Entrega el número bins")
bi = int(input())
```

```
X_dict = OrderedDict()
x_old ={}
x_new ={}
for i in range(len(x)):
    X_dict[i]= x[i]
    x_old[i]= x[i]

x_dict = sorted(X_dict.items(), key = lambda x: x[1])

binn = []
avrg = 0
i = 0
k = 0
num_of_data_in_each_bin = int(math.ceil(len(x)/bi))

for g, h in X_dict.items():
    if(i<num_of_data_in_each_bin):
        avrg = avrg + h
        i = i + 1
    elif(i == num_of_data_in_each_bin):
        k = k + 1
```

```
        i = 0
        binn.append(round(avrg /
num_of_data_in_each_bin, 3))
        avrg = 0
        avrg = avrg + h
        i = i + 1
rem = len(x)% bi
if(rem == 0):
    binn.append(round(avrg / num_of_data_in_each_bin,
3))
else:
    binn.append(round(avrg / rem, 3))
i = 0
j = 0
for g, h in X_dict.items():
    if(i<num_of_data_in_each_bin):
        x_new[g]= binn[j]
        i = i + 1
    else:
        i = 0
        j = j + 1
        x_new[g]= binn[j]
        i = i + 1
```

```
print("El número de datos en cada bin")

print(math.ceil(len(x)/bi))


for i in range(0, len(x)):

    print('indice {2} valor anterior {0} valor nuevo
{1}'.format(x_old[i], x_new[i], i))
```

Detección y filtrado de Outliers

Los **outliers** en nuestro **dataset** serán los valores que se "escapan al rango en donde se concentran la mayoría de muestras". Según Wikipedia son *las muestras que están distantes de otras observaciones*.

Detección de Outliers

¿Y por qué nos interesa detectar estos Outliers? Por qué pueden afectar considerablemente a los resultados que pueda obtener un modelo de Machine Learning… Para mal… o para bien! Por eso hay que detectarlos, y tenerlos en cuenta. Por ejemplo en Regresión Lineal o algoritmos de Ensamble puede tener un impacto negativo en sus predicciones.

Los Outliers pueden significar varias cosas:

1. **ERROR**: Si tenemos un grupo de "edades de personas" y tenemos una persona con 160 años, seguramente sea un error de carga de datos. En este caso, la detección de outliers nos ayuda a detectar errores.
2. **LIMITES**: En otros casos, podemos tener valores que se escapan del "grupo medio", pero queremos mantener el dato modificado, para que no perjudique al aprendizaje del modelo de ML.

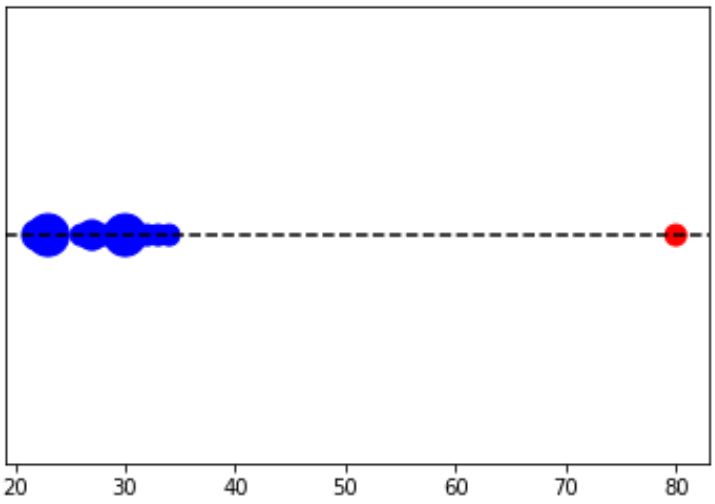
3. **Punto de Interés**: puede que sean los casos “anómalos” los que queremos detectar y que sean nuestro objetivo (y no nuestro enemigo!)

Muchas veces es sencillo identificar los outliers en gráficas. Veamos ejemplos de Outliers en 1, 2 y 3 dimensiones.

Outtliers en 1 dimensión

Si realizáramos una sola variable, por ejemplo “edad”, veremos donde se concentran la mayoría de muestras y los posibles valores “extremos”. Pasemos a un ejemplo en Python!

```
import matplotlib.pyplot as plt
import numpy as np

edades = np.array([22,22,23,23,23,23,26,27,27,28,30,30,30,30,31,32,33,34,80])
edad_unique, counts = np.unique(edades, return_counts=True)

sizes = counts*100
colors = ['blue']*len(edad_unique)
colors[-1] = 'red'

plt.axhline(1, color='k', linestyle='--')
plt.scatter(edad_unique,np.ones(len(edad_unique)),s=sizes,color=colors)
plt.yticks([])

plt.show()
```



En azul los valores donde se concentra la mayoría de nuestras filas. En rojo un outlier, ó “valor extremo”.

En el código, importamos librerías, creamos un array de edades con Numpy y luego contabilizamos las ocurrencias.

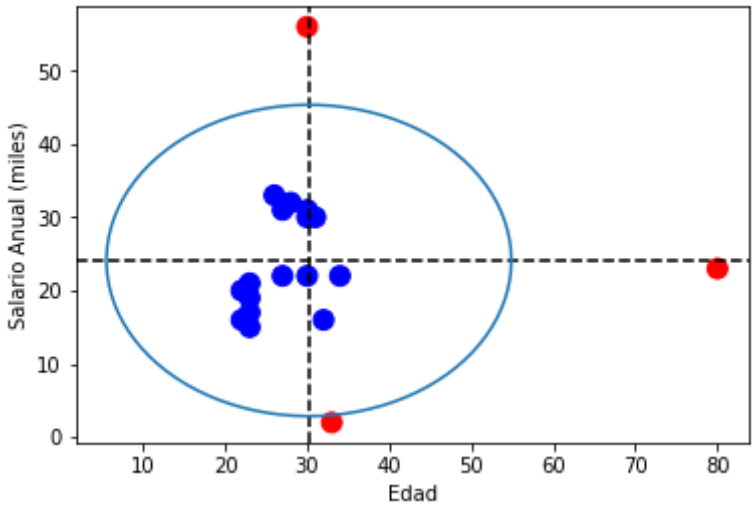
Al graficar vemos donde se concentran la mayoría de edades, entre 20 y 35 años. Y una muestra aislada con valor 80.

Outtliers en 2 dimension

Ahora supongamos que tenemos 2 variables: edad e ingresos. Hagamos una gráfica en 2D. Además, usaremos una fórmula para trazar un círculo que delimitará los valores outliers: Los valores que superen el valor de la “media más 2 desvíos estándar” (el área del círculo) quedarán en rojo.

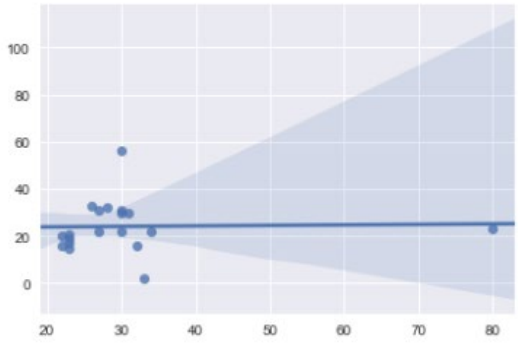
```
from math import pi

salario_anual_miles =
np.array([16,20,15,21,19,17,33,22,31,32,56,30,22,31,30,16,2,22,23])

media = (salario_anual_miles).mean()
std_x = (salario_anual_miles).std()*2
media_y = (edades).mean()
std_y = (edades).std()*2
```

```
colors = ['blue']*len(salario_anual_miles)
for index, x in enumerate (salario_anual_miles):
         if abs(x-media) > std_x:
                  colors[index] = 'red'

for index, x in enumerate(edades):
         if abs(x-media_y) > std_y:
                  colors[index] = 'red'

plt.scatter(edades, salario_anual_miles, s=100, color=colors)
plt.axhline(media, color='k', linestyle= '--')
plt.axvline(media_y, color='k',  linestyle= '--')

v = media   # posición-y del centro
u = media_y # posición-x del centro
b = std_x   # radio en el eje-y
a = std_y   # radio en el eje-x

t = np.linspace(0, 2*pi, 100)
plt.plot(u+a*np.cos(t), v+b*np.sin(t))
plt.xlabel('Edad')
plt.ylabel('Salario Anual (miles)')
plt.show()
```

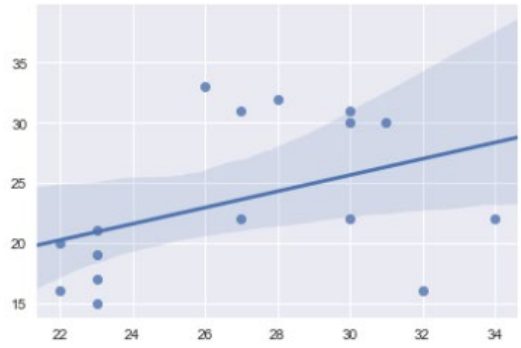


Dentro del círculo azul, los valores que están en la media y en rojo los outliers: 3 valores que superan en más de 2 veces el desvío estándar.

Veamos -con la ayuda de seaborn-, **la línea de tendencia de la misma distribución** con y sin outliers:



CON OUTLIERS: La línea de tendencia se mantiene plana sobre todo por el outlier de la edad



SIN OUTLIERS: Al quitar los outliers la tendencia empieza a tener pendiente

Con esto nos podemos dar una idea de qué distinto podría resultar entrenar un modelo de Machine Learning con o sin esas muestras anormales.

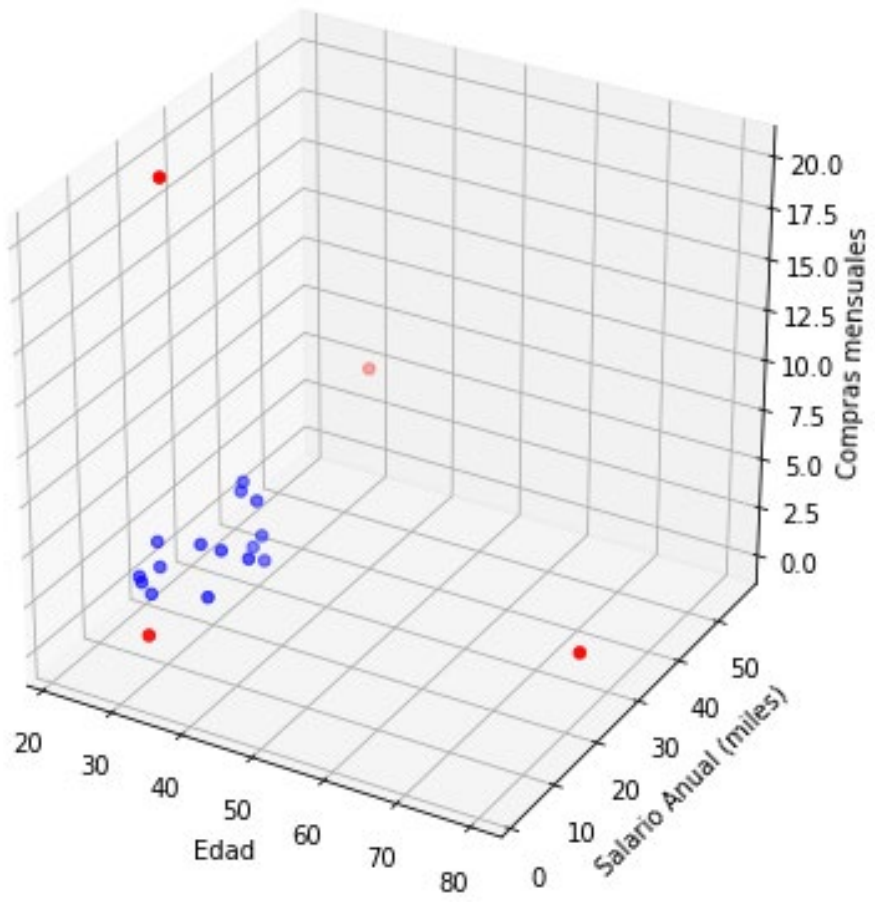
Outtliers en 3D

Vamos viendo que algunas de las muestras del **dataset** inicial van quedando fuera!

¿Qué pasa si añadimos una 3ra dimensión a nuestro **dataset**? Por ejemplo, la dimensión de “compras por mes” de cada usuario.

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(7,7))
ax = fig.gca(projection='3d')

compras_mes = np.array([1,2,1,20,1,0,3,2,3,0,5,3,2,1,0,1,2,2,2])

media_z = (compras_mes).mean()
std_z = (compras_mes).std()*2

for index, x in enumerate(compras_mes):
         if abs(x-media_z) > std_z:
                  colors[index] = 'red'

ax.scatter(edades, salario_anual_miles, compras_mes, s=20, c=colors)
plt.xlabel('Edad')
plt.ylabel('Salario Anual (miles)')
ax.set_zlabel('Compras mensuales')
```

```
plt.show()
```



Vemos en 3 dimensiones que hay valores que escapan a la <<**distribución normal**>>. Valores atípicos en rojo.

En el caso de las compras mensuales, vemos que aparece un nuevo "punto rojo" en el eje Z. Debemos pensar si es un usuario que queremos descartar o que por el contrario, nos interesa analizar.

Librería Python para detección de Outliers

En el código utilicé una medida conocida para la detección de outliers que puede servir: **la media de la distribución más 2 sigmas como frontera**. Pero existen otras estrategias para delimitar outliers.

Una librería muy recomendada es PyOD. Posee diversas estrategias para detectar Outliers. Ofrece distintos algoritmos, entre ellos Knn que tiene mucho sentido, pues analiza la cercanía entre muestras, PCA, Redes Neuronales, veamos cómo utilizarla en nuestro ejemplo.

```
!pip install pyod #instala la libreria

from pyod.models.knn import KNN

import pandas as pd


X = pd. DataFrame(data={'edad':edades, 'salario':
salario_anual_miles, 'compras': compras_mes})

clf = KNN(contamination=0.18)

clf.fit(X)

y_pred = clf.predict(X)

X[y_pred == 1]
```

	compras	edad	salario
3	20	23	21
10	5	30	56
16	2	33	2
18	2	80	23

La librería PyOd detecta los registros anómalos.

Para problemas en la vida real, con múltiples dimensiones conviene apoyarnos en una librería como esta que nos facilitará la tarea de detección y limpieza/transformación del dataset.

Permutación de la data y muestreo aleatorio

Las permutaciones se usan en probabilidad y estadística para determinar el número de formas en las que una lista de elementos puede ser arreglada. Python no incluye funciones de permutación integradas, sin embargo el cálculo puede realizarse mediante el "Módulo de itertools". Una función de "permutaciones" está incluida en este módulo y puede calcular y listar el número de permutaciones dado un "conjunto" de valores y una longitud de permutación de "r".

Paso 1

Importa el módulo "itertools":

```
import itertools
```

Paso 2

Asigna todos los valores del conjunto a una variable. Supón que necesitas determinar el número de formas en las que 2 miembros de una hermandad de 5 estudiantes pueden ser elegidos en posiciones como Vicepresidente y Presidente. Asumiendo que sus nombres son Angela, Cindy, Jan, Marsha y Beyonce, tendrías que escribir el siguiente comando:

```
permutation_set_variable = ['Angela', 'Cindy', 'Jan',
'Marsha', 'Beyonce']
```

Paso 3

Ejecuta la función "permutations" (permutaciones) de la siguiente forma:

```
itertools.permutations(set, r)
```

Paso 4

Sustituye "set" con la variable que contenga los valores y "r" con la longitud requerida. Según el ejemplo tendrías que escribir lo siguiente:

```
itertools.permutations(permutation_set_variable, 2)
```

Muestreo aleatorio

El método de muestreo más básico es el muestreo aleatorio, en el que simplemente se selecciona un conjunto de subconjunto de los datos de forma aleatoria. A pesar de ello es uno de los métodos más utilizados debido a su simplicidad. Además de ser válido en la mayoría de los problemas.

Los **DataFrames** de Pandas incluyen el método **sample** que nos permite implementar el muestreo aleatorio en Python. Así para obtener n muestras aleatorias de un **DataFrame** solamente se tiene que escribir

```
sample_df = df.sample(n)
```

Al igual que la mayoría de métodos que usan el generador de números aleatorios se puede fijar la semilla mediante la propiedad random_state. Algo que debemos utilizar siempre que nuestros análisis necesiten ser reproducibles, es decir, obtengamos los mismos valores al repetir el ejercicio con los mismos conjuntos de datos.

Al trabajar con datos puede que sea necesario realizar una muestra de un conjunto de datos. Esto puede ser porque la totalidad de los datos disponibles sea inmanejable o porque exista un desequilibrio en las clases de los datos. En estos es necesario recurrir al muestreo de los datos. Algunas de las técnicas más utilizadas de muestreo en Python se muestran a continuación.

Muestreo estratificado

Supongamos que en nuestro conjunto de datos tengamos algunas características o conjunto de características cuya proporción necesitamos que se mantenga en el conjunto muestreo. Por ejemplo, la proporción de hombres y mujeres. En este caso lo que necesitamos hacer es un muestreo estratificado.

Pandas no incluye un método en sí para realizar un muestreo estratificado, pero se puede utilizar **train_test_split** de **Scikit-learn** para obtener este muestreo en Python. Este método tiene la propiedad stratify con la que se puede definir un conjunto de variables para estratificar. Así para obtener una muestra del 10% de un **DataFrame** solamente debemos escribir el comando

```
X_train, X_test, y_train, y_test =
train_test_split(X, y, stratify = y, test_size = 0.1)
```

En donde y es la variable sobre la que se desea estratificar

Undersampling

En muestras desbalanceadas, donde existen clases que contiene más registros que otros y es necesario balancear los registros es necesario usar un muestreo tipo "**undersampling**" u "**oversampling**" (el cual veremos en la próxima sección). Por ejemplo, un problema de predicción de fraude donde tenemos 1 caso de fraude por cada 1000 registros válidos. En el muestreo tipo "**undersampling**" se selecciona un subconjunto de la clase mayoritaria (en nuestro ejemplo los registros válidos) y todos los registros de la clase mayoritaria. Algo que se puede hacer fácilmente de forma aleatoria. Simplemente se tiene que separar las clases en dos **DataFrames** en base al valor y emplear el método **sample** como ya hemos visto y combinar en un nuevo **DataFrame**:

```
df_true = df[df['y'] == True]
df_false = df[df['y'] == True]
df_sample = pd.concat([df_true,
df_false.sample(df_true.shape[0])])
```

Manipulación string

Saber cómo manipular cadenas de caracteres juega un papel fundamental en la mayoría de las tareas de procesamiento de texto.

Cadena de caracteres

Si has estado expuesto antes a otro lenguaje de programación, sabrás que necesitas *declarar* o *escribir* variables antes de que puedas almacenar cualquier cosa en ellas. Esto no es necesario cuando trabajas con cadenas de caracteres en Python. Podemos crear una cadena de caracteres simplemente encerrando contenido entre comillas después de un signo de igual (=).

```
mensaje = "Hola Mundo"
```

Operadores de cadenas: Adición y multiplicación

Una cadena de caracteres es un objeto que consiste precisamente en una serie de signos o caracteres. Python sabe cómo tratar un número de representaciones poderosas y de propósito general, incluidas las cadenas de caracteres. Una forma de manipular cadenas de caracteres es utilizar *operadores de cadenas de caracteres*. Dichos operadores se representan con símbolos que asociamos a las matemáticas, como +, -, *, / y =. Estos signos realizan acciones similares a sus contrapartes matemáticas cuando se usan con las cadenas de caracteres, aunque no iguales.

Concatenar

Este término significa juntar cadenas de caracteres. El proceso de concatenación se realiza mediante el operador de suma (+). Ten en cuenta que debes marcar explícitamente dónde quieres los espacios en blanco y colocarlos entre comillas.

En este ejemplo, la cadena de caracteres “mensaje1” tiene el contenido “***Hola Mundo***”

```
mensaje1 = "Hola" + " " + "Mundo"

print(mensaje1)
```

```
-> Hola Mundo
```

Multiplicar

Si quieres varias copias de una cadena de caracteres utiliza el operador de multiplicación (*). En este ejemplo, la cadena de caracteres ***mensaje2a*** lleva el contenido “Hola” tres veces, mientras que la cadena de caracteres ***mensaje2b*** tiene el contenido “Mundo”. Ordenemos imprimir las dos cadenas.

```
mensaje2a = "Hola "*3
mensaje2b = "Mundo"

print(mensaje2a + mensaje2b)
```

```
-> Hola Hola Hola Mundo
```

Añadir

¿Qué pasa si quieres añadir material de manera sucesiva al final de una cadena de caracteres? El operador especial para ello es compuesto (+=).

```
mensaje3 = "Hola"
mensaje3 += " "
mensaje3 += "Mundo"

print(mensaje3)
```

```
-> Hola Mundo
```

Métodos para cadenas de caracteres: buscar, cambiar

En adición a los operadores, Python trae preinstalado docenas de métodos que te permiten hacer cosas con las cadenas de caracteres. Solos o en combinación, los métodos pueden hacer casi todo lo que te imagines con las cadenas de caracteres. Puedes usar como referencia la lista de métodos de cadenas de caracteres (*String Methods*) en el sitio web de Python, que incluye información de cómo utilizar correctamente cada uno. Para asegurar que tengas una comprensión básica de métodos para cadenas de caracteres, lo que sigue es una breve descripción de los utilizados más comúnmente.

Extensión

Puedes determinar el número de caracteres en una cadena utilizando el método **len**. Acuérdate que los espacios en blanco cuentan como un carácter.

```
mensaje4 = "Hola" + " " + "Mundo"

print(len(mensaje4))
```

```
-> 10
```

Encontrar

Puedes buscar una sub-cadena en una cadena de caracteres utilizando el método **find** y tu programa te indicará el índice de inicio de la misma. Esto es muy útil para procesos que veremos más adelante. Ten en mente que los índices están numerados de izquierda a derecha y que el número en el que se comienza a contar la posición es el 0, no el 1.

```
mensaje5 = "Hola Mundo"
mensaje5a = mensaje5.find("Mundo")

print(mensaje5a)
```

```
-> 5
```

Si la sub-cadena no está presente el programa imprimirá el valor -1.

```
mensaje6 = "Hola Mundo"
mensaje6a = mensaje5.find("ardilla")

print(mensaje6a)
```

```
-> -1
```

Minúsculas

A veces es útil convertir una cadena de caracteres a minúsculas. Para ello se utiliza el método **lower**. Por ejemplo, al uniformar los caracteres permitimos que la computadora reconozca fácilmente que “Algunas Veces” y “algunas veces” son la misma frase.

```
mensaje7 = "HOLA MUNDO"
mensaje7a = mensaje7.lower()

print(mensaje7a)
```

```
-> hola mundo
```

Convertir las minúsculas en mayúsculas se logra cambiando .**lower**() por **upper**().

Reemplazar

Si necesitas cambiar una sub-cadena de una cadena se puede utilizar el método replace.

```
mensaje8 = "HOLA MUNDO"
mensaje8a = mensaje8.replace("L", "pizza")

print(mensaje8a)
```

```
-> HOpizzaA MUNDO
```

Contar

Si quieres cortar partes que no quieras del principio o del final de la cadena de caracteres, lo puedes hacer creando una sub-cadena. El mismo tipo de técnica te permite separar una cadena muy larga en componentes más manejables.

```
mensaje9 = "Hola Mundo"
mensaje9a = mensaje9[1:8]

print(mensaje9a)
```

```
-> ola Mun
```

Puedes sustituir las variables por números enteros como en este ejemplo:

```
mensaje9 = "Hola Mundo"
startLoc = 2
endLoc = 8
mensaje9b = mensaje9[startLoc:endLoc]

print(mensaje9b)
```

```
-> la Mun
```

Esto hace mucho más simple usar este método en conjunción con el método find como en el próximo ejemplo, que busca la letra “d” en los seis primeros caracteres de “Hola Mundo” y correctamente nos dice que no se encuentra ahí (-1). Esta técnica es mucho más eficaz en cadenas largas -documentos enteros, por ejemplo. Observa que la ausencia de un número entero antes de los dos puntos significa que queremos empezar desde el principio de la cadena. Podemos usar la misma técnica para decirle al programa que pase hasta el final de la cadena de caracteres dejando vacío después de los dos puntos. Y recuerda que la posición del índice empieza a contar desde 0, no desde 1.

```
mensaje9 = "Hola Mundo"

print(mensaje9[:5].find("d"))
```

```
-> -1
```

Hay muchos más, pero los métodos para cadenas de caracteres anteriores son un buen comienzo. Fíjate que en el ejemplo anterior utilizamos corchetes en vez de paréntesis. Esta diferencia en los símbolos de la *sintaxis* es muy importante. Los paréntesis en Python son utilizados generalmente para *llevar un argumento* a una función. De tal manera que cuando vemos algo como:

```
print(len(mensaje7))
```

Quiere decir que se lleva la cadena de caracteres “mensaje7” a la función len y entonces enviar el valor resultante de esa función a la declaración print para ser impresa. Una función puede ser llamada sin un argumento, pero de todas formas tienes que incluir un par de paréntesis

vacíos después del nombre de la función. Vimos un ejemplo de ello también.

```
mensaje7 = "HOLA MUNDO"
mensaje7a = mensaje7.lower()

print(mensaje7a)
```

```
-> Hola Mundo
```

Esta declaración le dice a Python que aplique la función **lower** a la cadena **mensaje7** y guarde el valor resultante en la cadena **mensaje7a**.

Los corchetes sirven para propósitos diferentes. La cadena es una secuencia de caracteres; así que si quieres acceder al contenido de la cadena a partir de su posición en la secuencia, tienes que indicarle a Python un lugar en la secuencia. Eso es lo que hacen los corchetes: señalan el lugar del principio y del final de la secuencia, tal y como vimos en el método cortar.

Secuencia de escape.

¿Qué haces cuando necesitas incluir comillas en una cadena de caracteres? No quieres que el intérprete de Python se equivoque y piense que la cadena termina en donde se encuentre una comilla. En Python puedes poner una barra invertida (\) enfrente de la comilla para que no acabe ahí la cadena. Esto es conocido como secuencia de escape.

```
print('\"')
```

```
-> "
```

```
print('El programa imprime \"Hola Mundo\"')
```

```
-> El programa imprime "Hola Mundo"
```

Otras dos secuencias de escape te permiten incluir marcas de tabulación (t) y saltos de línea (n):

```
print('Hola\tHola\tHola\nMundo')
```

```
-> Hola Hola Hola
Mundo
```

El método DataFrame.apply

Los dataframe tienen un método con el mismo nombre que el método **apply** de las series, **pandas.DataFrame.apply**, pero con funcionalidad diferente pues, en el caso de los dataframe, se aplica a lo largo de un eje del dataframe. Esto quiere decir que el argumento de entrada de la función a utilizar no va a ser un simple escalar, sino una serie cuyo índice va a ser el índice de filas del dataframe (si la función se aplica al eje 0) o el índice de columnas del dataframe (si la función se aplica al eje 1). El resultado del método también será una serie que estará formada por los valores calculados.

Por ejemplo, si tenemos el siguiente dataframe con las ventas de los productos *A*, *B*, *C* y *D* a lo largo de los meses de enero, febrero y marzo:

```
ventas = pd.DataFrame({'A': [3, 3, 1],
                       'B': [1, 5, 2],
                       'C': [3, 7, 2],
                       'D': [7, 2, 3]},
                      index = ['Ene', 'Feb', 'Mar'])

ventas
```

	A	B	C	D
Ene	3	1	3	7
Feb	3	5	7	2
Mar	1	2	2	3

...podríamos estar interesados en calcular el rango en el que se mueven las ventas, es decir, la diferencia entre el mayor y el menor valor de ventas. Para ello, sabiendo que dicho rango se va a aplicar a una fila o a una columna -es decir, a una serie-, definimos la siguiente función:

```
def rango(s):
      return max(s) - min(s)
```

Esta función acepta un iterable y devuelve la diferencia entre el valor máximo y el mínimo.

Ahora podemos aplicar esta función a nuestro dataframe de ventas. Por defecto se va a aplicar al eje 0 (eje vertical):

```
ventas.apply(rango)
```

```
A    2
B    4
C    5
D    5
dtype: int64
```

Si nos fijamos en la columna A, el valor máximo es 3 y el mínimo es 1, de forma que su diferencia es 2, tal y como se muestra en el resultado del método apply.

Si aplicamos el método a lo largo del eje 1 (eje horizontal), obtendremos la diferencia entre el mayor y el menor valor de cada fila:

```
ventas.apply(rango, axis = 1)
```

```
Ene    6
Feb    5
Mar    2
dtype: int64
```

Referencias

[1] Valores Perdidos
https://www.youtube.com/watch?v=_i-c80qYqbs&t=22s

[2] Imputación de datos
https://ichi.pro/es/serie-de-limpieza-de-datos-con-python-40387192062664

[3] Imputaciones k-NN
https://www.youtube.com/watch?v=FHHuo7xEeo4
https://www.merkleinc.com/es/es/blog/algoritmo-knn-modelado-datos

[4] MultiIndex / Indexación avanzada
https://pandas.pydata.org/pandas-docs/dev/user_guide/advanced.html#

[5] Permutación de data.
https://www.cienciadedatos.net/documentos/pystats03-test-permutacion-python.html