Estadística descriptiva para el mantenimiento industrial con PYTHON

Eduardo Segundo Hernández-Dávila
César Marcelo Gallegos-Londoño
Félix Antonio García-Mora

CIDE EDITORIAL

Estadística descriptiva para el mantenimiento industrial con PYTHON

Estadística descriptiva para el mantenimiento industrial con PYTHON

Autores

EDUARDO SEGUNDO HERNÁNDEZ-DÁVILA
CÉSAR MARCELO GALLEGOS-LONDOÑO
FÉLIX ANTONIO GARCÍA-MORA

espoch

Dirección de Publicaciones

Estadística descriptiva para el mantenimiento industrial con Python

Centro de Investigación y Desarrollo Ecuador
Tel.: + (593) 04 2037524
http.:/www.cidecuador.org

ISBN: 978-9942-636-55-3

https://doi.org/10.33996/cide.ecuador.EP2636553

Filiación:
Eduardo Segundo Hernández-Dávila
César Marcelo Gallegos-Londoño
Félix Antonio García-Mora
Escuela Superior Politécnica de Chimborazo

Dirección editorial: Lic. Pedro Misacc Naranjo, Msc.
Coordinación técnica: Lic. María J. Delgado
Diseño gráfico: Lic. Danissa Colmenares
Diagramación: Lic. Alba Gil
Fecha de publicación: diciembre, 2023

La presente obra fue evaluada por pares académicos experimentados en el área.

Catalogación en la Fuente

Estadística descriptiva para el mantenimiento industrial con Python / Eduardo Segundo Hernández-Dávila, César Marcelo Gallegos-Londoño y Félix Antonio García-Mora. - Ecuador: Editorial CIDE, 2023.

290 p.: incluye tablas, figuras; 17,6 x 25 cm.

ISBN: 978-9942-636-55-3

1. Estadística descriptiva 2. Mantenimiento industrial 3. Python

Índice

Capítulo 3
Medidas de tendencia central

Capítulo 4
Medidas de dispersión y posición

Capítulo 5
Análisis de la forma de la distribución

Capítulo 6
Manejo de datos faltantes y atípicos

Capítulo 7
Elaboración de gráficos estadísticos

Capítulo 8
Casos prácticos

Prólogo

El presente libro ha sido redactado como un aporte a la carrera de Mantenimiento Industrial en la asignatura de Estadística, con una fusión entre las TIC´s y las matemáticas, el mismo que sirve para cualquier área ya sea Mecánica, Automotriz, Ingeniería industrial entre otras, que necesiten trabajar de la mano con la tecnología.

Para el seguimiento de este libro se recomienda tener un conocimiento básico de programación y manejo del software Python, pues el libro consiste en enseñar como desarrollar temas de Estadística Descriptiva con la ayuda de las TIC´s, además como objetivo los autores pretenden contribuir con la formación del futuro Ingeniero de Mantenimiento Industrial, el lector podrá ser capaz de manejar, manipular, interpretar y presentar datos para la toma de decisiones en base a los posibles datos que tendrán que analizar en un futuro no muy lejano.

Los temas estadísticos abordados van acompañados de ejemplos prácticos a través de la implementación del software de programación Python, partiendo desde la instalación de Python, avanzando con la instalación de librerías, para proceder a resolver problemas de estadística descriptiva aplicados al Mantenimiento Industrial y finalmente culminar con un ejemplo práctico paso a paso como ejemplo de lo desarrollado en capítulos anteriores.

Los temas tratados son: instalación de Python hasta el proceso de la instalación de librerías necesarias para la manipulación de datos, se continúa con la recolección y organización de los datos (introduciendo al lector desde conceptos básicos como son población, muestra, tipos de datos, elaboración de tablas de frecuencias), sigue luego con las medidas de tendencia central, medidas de dispersión y posición, análisis de distribución, el manejo de los datos faltantes, para una mejor interpretación elaboración de gráficos estadísticos y para reforzar los temas tratados casos prácticos.

En cada uno de los capítulos, los autores presentan ejemplos que muestran el manejo de código necesario para la obtención del tema a tratar con el fin de incentivar al lector de implementar

una herramienta informática para el desarrollo de cada uno de los temas planteados así como la explicación de cada uno de los cálculos a desarrollar para obtener la información pertinente, así con esta disciplina motivar al lector a continuar con nuevas herramientas informáticas el desarrollo de la estadística descriptiva.

Cada uno de los capítulos cuenta con ejemplo que si el texto se lo maneja en digital podrá fácilmente ser cargado a Python para su aplicación y ejecución, los problemas planteados enfocados al área del mantenimiento producto del nivel de experiencia en el área del mantenimiento de los autores y así poder profundizar en algún área de interés, al finalizar el texto se presentan ejercicios propuestos.

"Los datos se están convirtiendo en la nueva materia prima de los negocios".

Craig Mundie

Vanessa L. Valverde González
Ingeniera en Sistemas Informáticos
Facultad de Mecánica- CIMANT -ESPOCH

Introducción

La estadística descriptiva con *Python* se refiere a la aplicación de técnicas y herramientas en el lenguaje de programación *Python* para explorar, resumir y visualizar conjuntos de datos con el objetivo de obtener una comprensión completa de sus características y patrones fundamentales. Este tipo de análisis se enfoca en proporcionar una descripción detallada de los datos sin necesariamente buscar establecer relaciones causales o llevar a cabo inferencias estadísticas.

En la ingeniería en general, desempeña un papel fundamental en la toma de decisiones en una variedad de disciplinas. Su importancia radica en que proporciona una comprensión sólida y objetiva de los datos, lo que permite a las personas tomar decisiones informadas y respaldadas por evidencia.

En el campo de la ingeniería de mantenimiento industrial, la estadística descriptiva ofrece información crucial para la planificación, el presupuesto y la toma de decisiones efectivas. Permite una gestión más eficiente de los materiales, repuestos, herramientas, personal y ejecución de actividades, lo que a su vez contribuye a la reducción de costos y tiempos de inactividad de una máquina o instalación industrial, así como a la mejora general de la eficiencia y la seguridad industrial.

La importancia de la estadística descriptiva en este contexto radica en varios aspectos clave como la optimización de recursos, la programación de mantenimiento, la reducción de tiempos de inactividad, el análisis de costos, la identificación de tendencias de rendimiento, la gestión de inventarios, la evaluación de la eficacia de estrategias de mantenimiento y la reducción de riesgos.

Por tales motivos en el Capítulo 1, se describe el procedimiento para instalar Python y VSCode, con las extensiones y librerías para que se pueda trabajar en el entorno de Jupyter Notebook. Seguidamente se sintetiza la configuración de los parámetros de las principales funciones de las librerías *NumPy*, *Pandas*, *Openpyxl*, *SciPy*, *Sklearn*, *Matplotlib* y *Seaborn* que se aplican en la estadística descriptiva.

En el capítulo 2 se explora los tipos de datos utilizados en el contexto del mantenimiento industrial, así como los métodos para importarlos a Python y la elaboración de las tablas de frecuencias mediante la aplicación de las librerías *NumPy* y *Pandas*.

Una vez que se cuenta con una base de datos tabulada, en el capítulo 3 se detalla los métodos de cálculo de las medidas de tendencia central de los valores independientes y agrupados, mediante el empleo de ecuaciones y funciones directas.

A continuación, en el capítulo 4, se desarrolla el cálculo de las medidas de dispersión y posición como el rango, desviación estándar, varianza, coeficiente de variación y cuantiles, muestrales y poblacionales, tanto de los valores independientes como de los agrupados.

En el capítulo 5, se realiza el análisis de la forma de la distribución, dentro de la cual se determina la normalidad de los datos mediante los métodos analítico y gráfico; posterior a eso, se calculan los indicadores de asimetría y curtosis a través de funciones directas de *Pandas*.

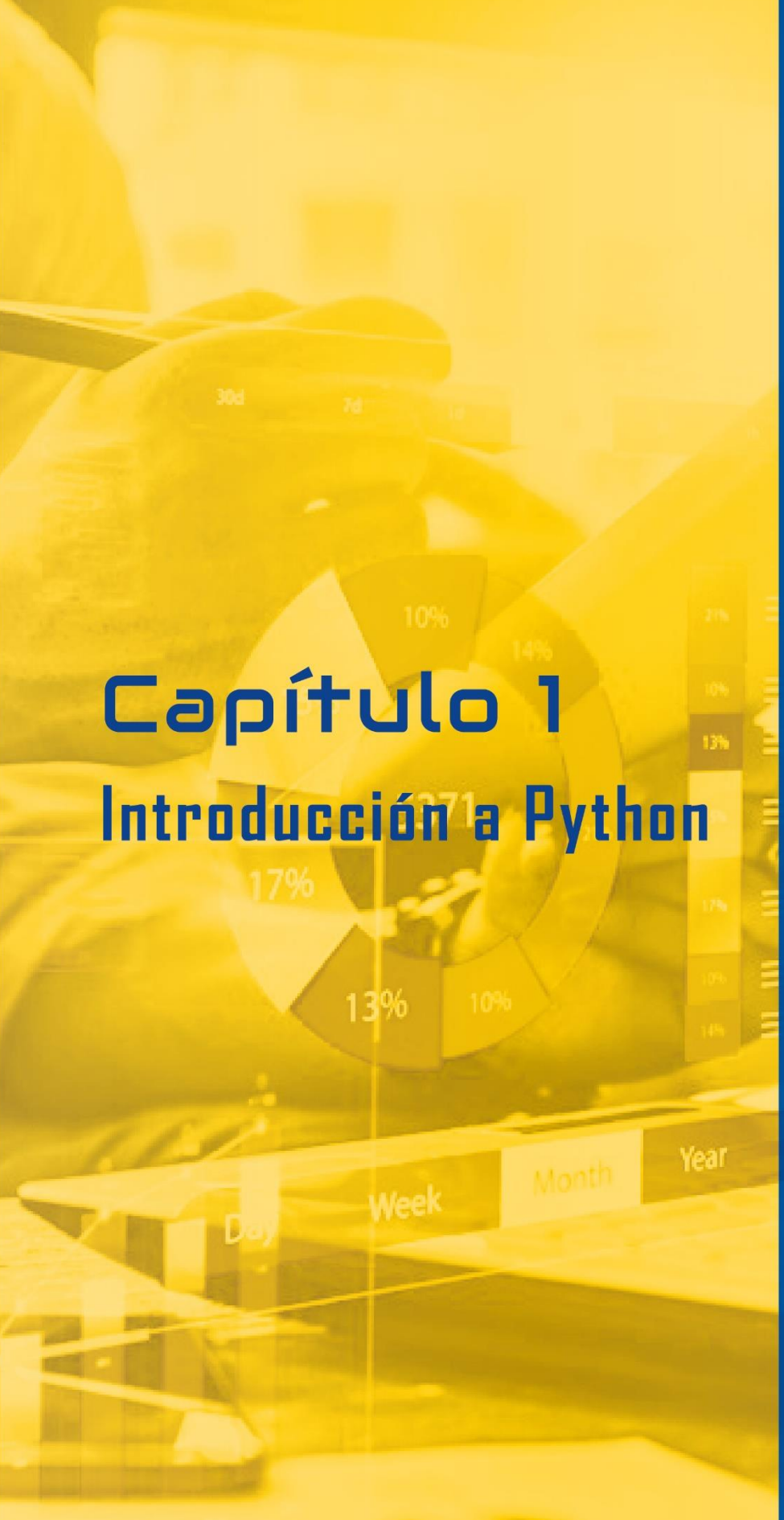
Seguidamente en el capítulo 6, se amplía sobre la eliminación de los registros con datos faltantes, así como también, la imputación simple y la imputación k-vecinos. A continuación, se expone sobre los métodos para la identificación y tratamiento de los valores atípicos.

Posteriormente en el capítulo 7, se detallan las configuraciones de las características de las funciones de las librerías Matplotlib y Seaborn para la elaboración de histogramas de frecuencias relativas, histogramas de frecuencias acumuladas, diagramas de cajas, diagramas de violín, gráficos de dispersión, diagramas de barras y diagramas de pastel.

Finalmente en el capítulo 8, se desarrolla una aplicación de la estadística descriptiva para el estudio del desgaste de un impulsor de una bomba sometida a la acción de un fluido abrasivo, para lo cual se inicia con la importación de una base de datos con las mediciones de los caudales de los últimos 4 años; para posteriormente realizar el tratamiento de los datos faltantes y atípicos, analizar la distribución de las muestras, explorar las medidas de tendencia central, de dispersión y de forma, y aplicar la linealización y consiguiente regresión lineal a las medias muestrales de los valores independientes para con ello estimar la vida residual del impulsor.

En los capítulos del 2 en adelante, se realizan aplicaciones de los temas tratados con datos y variables que se encuentran en el contexto del mantenimiento industrial; y de manera específica, los capítulos del 2 al 6 se desarrollan mediante la aplicación de una base de datos con mediciones de los caudales de una bomba centrífuga. Esta característica crea en el lector experimentado un ambiente familiar con el entorno industrial y en los estudiantes induce un panorama pragmático en el marco de las máquinas e instalaciones industriales, consiguiendo en los dos casos incentivar el estudio conjunto de la estadística descriptiva, el *Python* y la ingeniería de mantenimiento industrial, que es el objetivo principal de la presente obra.

Al final se proponen 22 ejercicios que abarcan la aplicación de la estadística descriptiva en el mantenimiento industrial mediante el empleo de *Python*, para profundizar al lector en los temas tratados y pueda reforzar las habilidades en la exploración y manejo de datos de las variables técnicas y de gestión que se generan en el ámbito del mantenimiento.



1

Capítulo 1

Introducción a Python

Capítulo

1

1. Introducción a Python

Python es un lenguaje de programación de uso general que se aplica en una amplia gama de campos, como la estadística, la ciencia de datos, el desarrollo web, el aprendizaje automático y la automatización. Es reconocido por su sintaxis clara y legible, lo que lo convierte en una opción atractiva tanto para principiantes como para desarrolladores experimentados. *Python* es un lenguaje de alto nivel y de interpretación, y ha ganado una gran popularidad gracias a su versatilidad y facilidad de uso (Coursera, 2023).

El entorno de desarrollo integrado incluido con la instalación estándar de *Python* es el IDLE; sin embargo, por su amplia aplicación en vario ámbitos, existen algunos entornos de desarrollo como *PyCharm*, *Spyder*, *Visual Studio Code* (*VSCode*), *Jupyter Notebook / JupyterLab*: *Jupyter*, *Google Colab*, *Sublime*

Text, *Django*, *Flask, Notepad++*, *Replit*, entre otros, cada uno con ventajas y desventajas y que son seleccionados de acuerdo con las necesidades específicas y preferencias personales de cada desarrollador (Prasad, 2023; Rootstack, 2022).

Para la aplicación de *Python* en la estadística descriptiva los autores han seleccionado como entorno de desarrollo a *VSCode*, porque ofrece una amplia gama de características y herramientas como el *IntelliSense* que proporciona sugerencias de código y completado automático para ayudar a escribir código más rápido y con menos errores, Depuración integrada que posibilita la depuración del código directamente desde el editor, y extensibilidad por lo que se pueden instalar una amplia variedad de extensiones y complementos muy útiles como la de *Jupiter* que permite a *VSCode* trabajar con los *Notebook/lab* (*.ipynb*) y en el entorno de las celdas de código y *Markdown* de *Jupiter* (Visual Studio Code, 2023b).

1.1 Instalación de *Python*

El primer paso es la instalación de *Python*, para lo cual se ingresa a la *web* de descarga oficial (Figura 1.1), mediante el

siguiente enlace https://www.python.org/downloads/, donde se debe seleccionar el instalador de acuerdo con el sistema operativo que tenga instalado en la computadora.

Figura 1.1

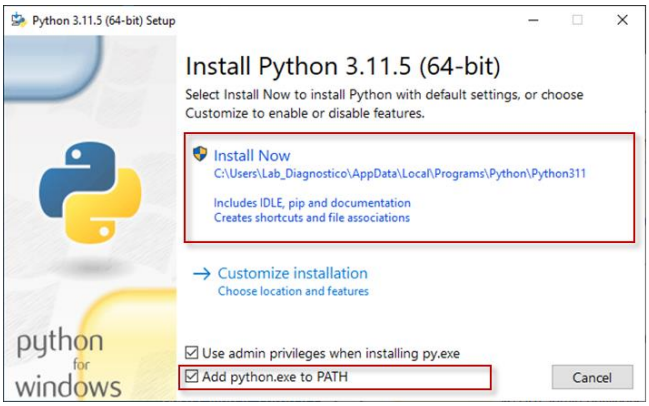
Página de descarga de Python.

Nota. Adaptado de (Python, 2023)**.**

Al ejecutar el archivo descargado para la instalación, se abre la ventana indicada en la Figura 1.2, donde de manera importante se debe seleccionar la opción "*Add python.exe to PATH*", posteriormente se presiona en "*Install Now*".

Figura 1.2

Ventana de instalación de Python.

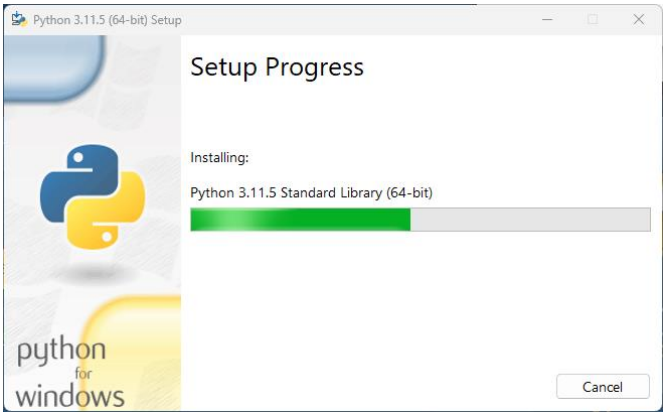


Nota. Los autores**.**

Seguidamente se abre la ventana mostrada en la Figura 1.3 que indica el progreso de la instalación.

Figura 1.3

Ventana del progreso de la instalación de Python.

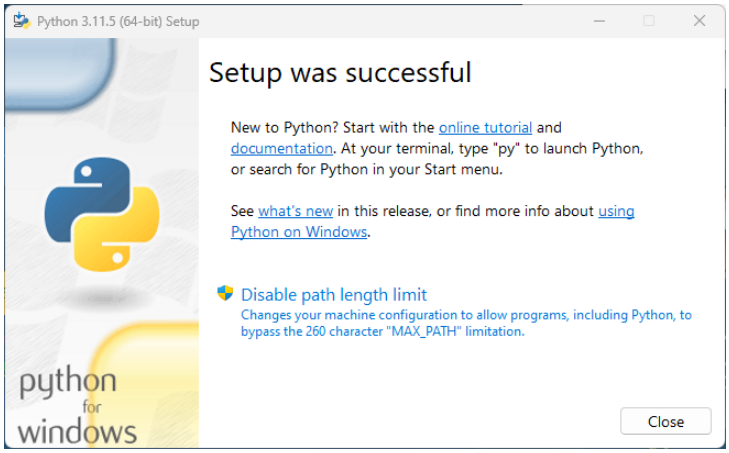


Nota. Los autores**.**

Al finalizar la instalación se muestra una ventana (Figura 1.4) indicando que la instalación fue satisfactoria (*Setup was successful*).

Figura 1.4

Ventana final de la instalación de Python.

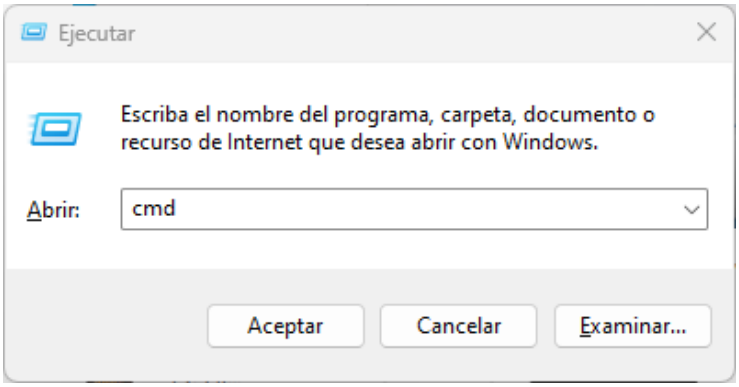


Nota. Los autores**.**

Para verificar que la instalación fue realmente satisfactoria se presiona las teclas combinadas "*Windows* + R", se escribe "*cmd*" (Figura 1.5) y se presiona "*intro*".

Figura 1.5

Ventana "Ejecutar" de Windows.



Nota. Los autores**.**

Siguiendo las indicaciones anteriores se abre la ventana de "Símbolo del Sistema" o "*Command Prompt*" donde se escribe "*Python --version*" y se presiona "*intro*". Con esto aparece la versión de *Python* instalada (Figura 1.6). Si se produce un error o no se presenta la versión de *Python* instalada, pude ser porque al inicio de la instalación no se seleccionó la opción "*Add python.exe to PATH*" indicada en la Figura 1.2.

Figura 1.6

Ventana de "Símbolo del Sistema" de Windows.

Nota. Los autores**.**

1.2 Instalación de *VSCode*

Para instalar *VSCode*, se ingresa a la web de descarga oficial (Figura 1.1), mediante el enlace https://code.visualstudio.com/download, donde se selecciona el instalador de acuerdo con el sistema operativo que tenga instalado en la computadora, de manera similar a como se hizo con la instalación de *Python*.

Figura 1.7

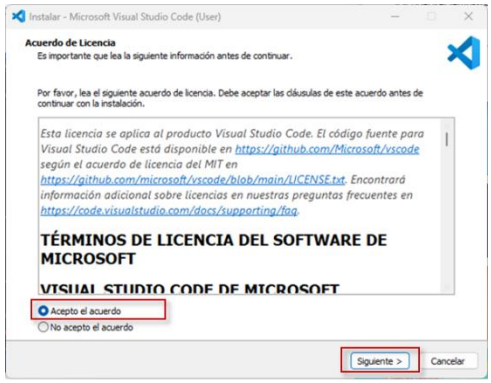
Página de descarga de VSCode.

Nota. Adaptado de (Visual Studio Code, 2023a)**.**

Luego de ejecutar el instalador se acepta el acuerdo de la licencia (Figura 1.8).

Figura 1.8

Ventana de "Acuerdo de licencia" de *VSCode*.

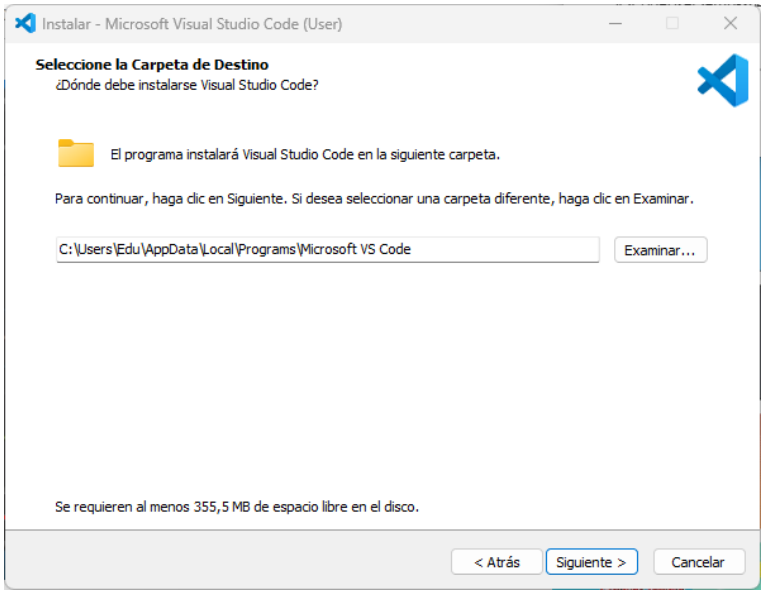


Nota. Los autores**.**

En el siguiente paso se debe indicar el directorio donde se desee que se carguen los archivos de la instalación (Figura 1.9). Se recomienda dejar el que sale por defecto.

Figura 1.9

Ventana para seleccionar el directorio de instalación de VSCode.

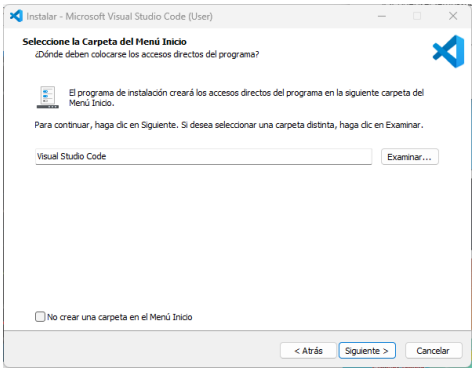


Nota. Los autores**.**

En la siguiente ventana se recomienda presionar el botón "siguiente" sin cambiar la configuración predeterminada, salvo mejor criterio (Figura 1.10).

Figura 1.10

Ventana de selección de la carpeta del menú inicio de VSCode.

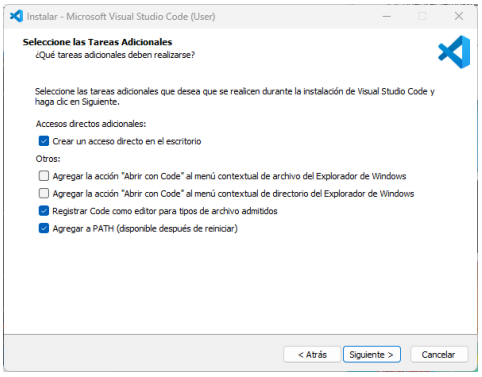


Nota. Los autores**.**

A continuación, se verifica que estén seleccionadas las 3 opciones indicadas en la Figura 1.11 y se presiona el botón "siguiente".

Figura 1.11

Ventana de selección de las tareas adicionales de la instalación de VSCode.

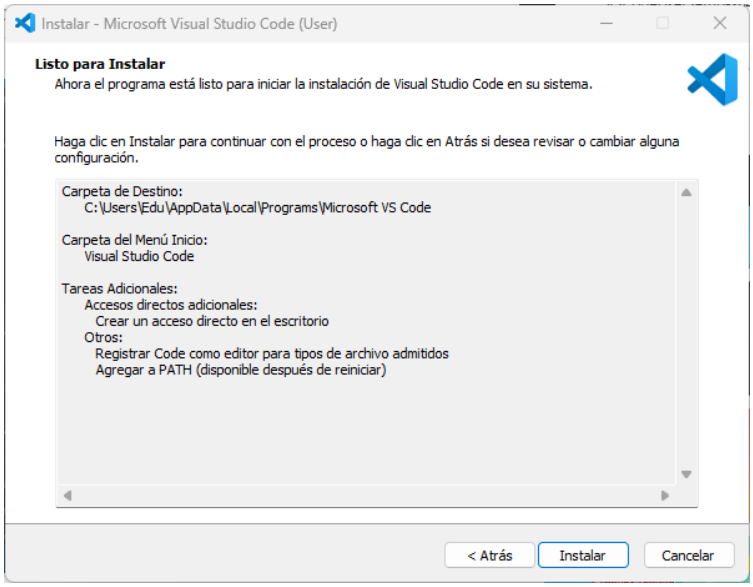


Nota. Los autores**.**

En el siguiente paso únicamente se presiona el botón “instalar” (Figura 1.12) e inmediatamente inicia la instalación.

Figura 1.12

Ventana previa a la instalación de VSCode.

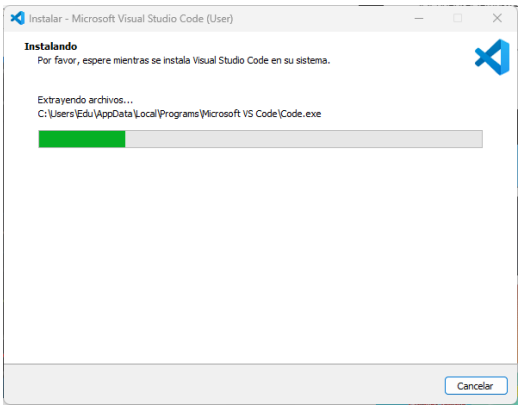


Nota. Los autores**.**

La ventana mostrada en la Figura 1.13 indica el progreso de la instalación, misma que dura unos pocos segundos.

Figura 1.13

Ventana del progreso de la instalación de VSCode.

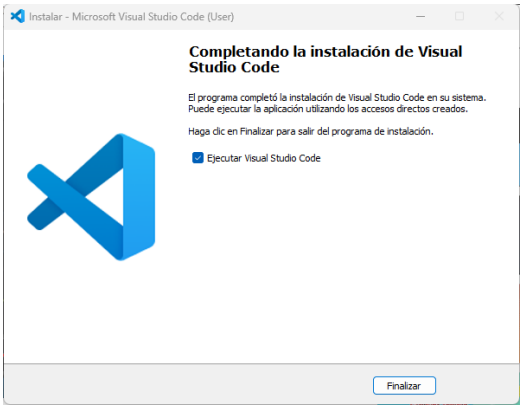


Nota. Los autores**.**

Como último paso, se presiona el botón "finalizar" con el visto en la opción "Ejecutar *Visual Studio Code*" (Figura 1.14) para que se abra la aplicación automáticamente.

Figura 1.14

Ventana final de la instalación de VSCode.



Nota. Los autores**.**

1.3 Instalaciones de las extensiones de *Python* y *Jupyter*

La instalación de las extensiones es imprescindible para poder utilizar *Python* con el entorno de *Jupyter Notebook/lab* en *VSCode*, y con beneficios como las sugerencias de código y completado automático, y la depuración del código directamente desde el editor.

Para esto, se presiona sobre el ícono de extensiones (barra vertical izquierda) o se presionan la combinación de teclas "*Ctrl* + *Shift* + X" y en el buscador se escribe la palabra "*python*". Luego de seleccionar la correspondiente a *Microsoft*, se presiona el botón azul "*Install*" del panel central de *VSCode*, tal como se puede observar en la Figura 1.15.

Figura 1.15

Instalación de la extensión de Python.

Nota. Los autores**.**

De manera similar, se escribe en el buscador de extensiones la palabra "*jupyter*". Luego de seleccionar la extensión de Microsoft, se presiona el botón azul "*Install*" del panel central de *VSCode*, tal como se puede observar en la Figura 1.16. La instalación termina en pocos segundos.

Figura 1.16

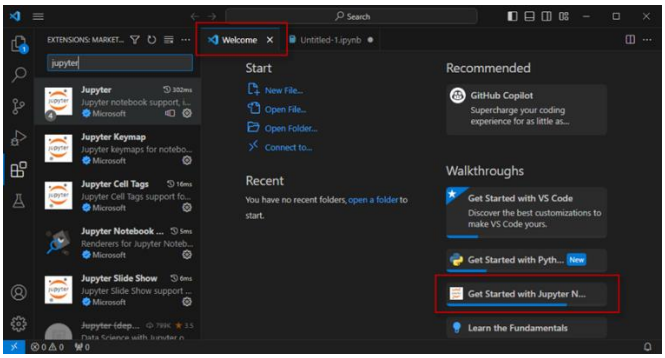
Instalación de la extensión de Jupyter.

Nota. Los autores**.**

Con las extensiones instaladas, se regresa a la pestaña "*welcome*" y en la sección "*Walkthroughs*" se presiona en "*Get Started with Jupyter Notebooks*" (Figura 1.17). Esta configura a *VSCode* para que al abrirse, aparezca de manera predeterminada las celdas "*Code*" / "*Markdown*" del entorno de *Jupyter Notebook*.

Figura 1.17

Configuración Get Started with Jupyter Notebooks.

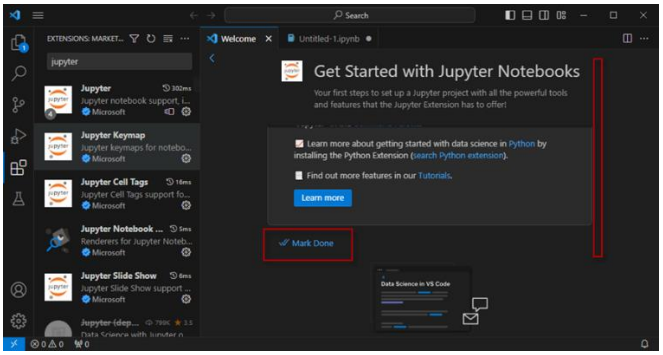


Nota. Los autores**.**

El panel central de *VSCode*, cambia para mostrar información sobre "*Get Started with Jupyter Notebooks*", luego desplácese con el "*scroll* vertical" hacia el final y presione en "*Mark Done*" (Figura 1.18).

Figura 1.18

Finalización de la configuración Get Started with Jupyter Notebooks.

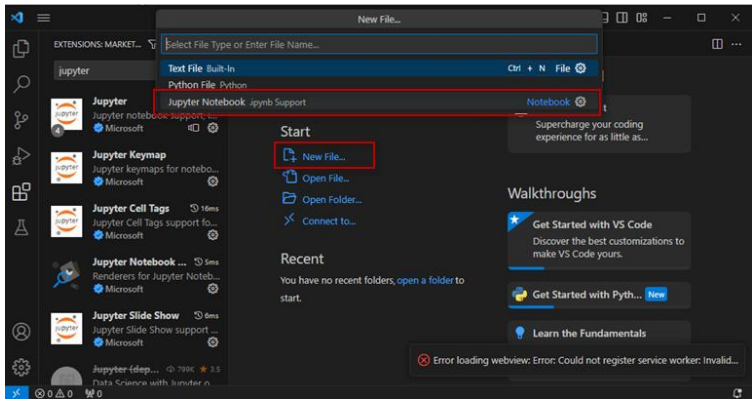


Nota. Los autores**.**

A continuación se presiona sobre "*New File…*" y en la ventana que se despliega, presione sobre la última opción "*Jupyter Notebook .ipynb Support*" (estas indicaciones abren normalmente un archivo *.ipynb* nuevo), como lo indica la Figura 1.19.

Figura 1.19

Apertura de un nuevo archivo de Jupyter Notebook.

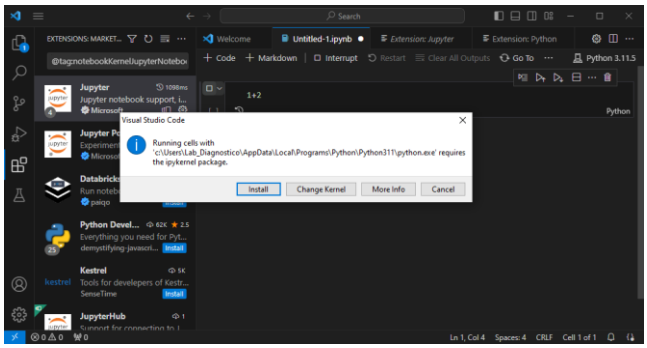


Nota. Los autores**.**

En la celda en blanco se escribe "1 + 2" y se presiona "*Ctrl + Alt + Intro*" y se abre una ventana emergente de *VSCode* (Figura 1.20), donde se sebe presionar en el botón "*Install*".

Figura 1.20

Instalación del Kernel de Python.

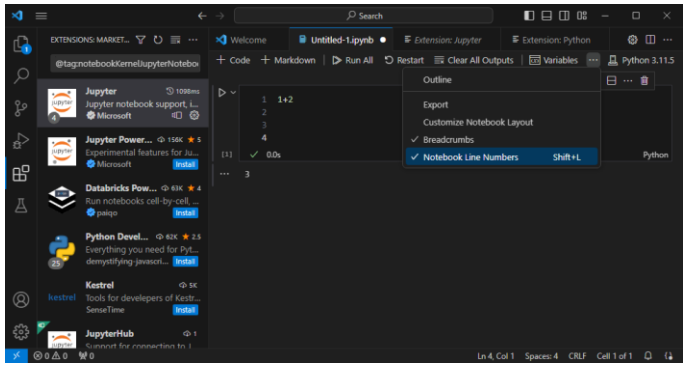


Nota. Los autores**.**

Por último, se presiona la combinación de teclas "*Shift* + L" o en la opción "*Notebook Line Numbers*" de la barra de herramientas de la extensión de *Jupiter* para habilitar el número de las líneas de las celdas (Figura 1.21).

Figura 1.21

Habilitación de la numeración de líneas de código.



Nota. Los autores**.**

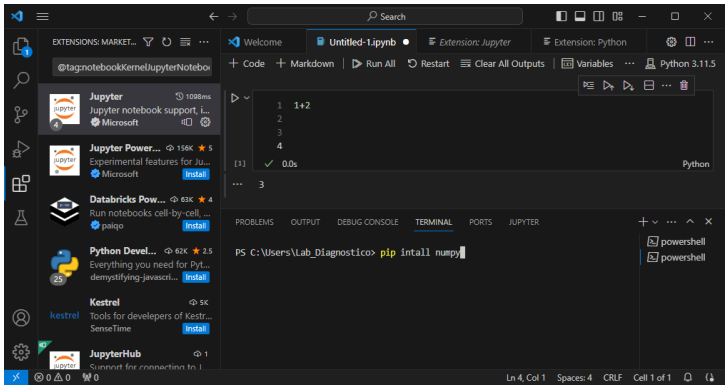
1.4 Instalación de librerías

Python cuenta con varias librerías empleadas en distintas áreas del conocimiento, de las cuales para la presente obra se van a utilizar *NumPy*, *Pandas*, *Openpyxl*, *SciPy*, *Sklearn*, *Matplotlib* y *Seaborn*, mismas que son ampliadas en el siguiente tema.

A continuación, se indica el proceso de instalación de las librerías, que se realiza abriendo una terminal dentro de *VSCode* presionando las teclas combinas "*Ctrl* + *Shift* + Ñ", en el cual se escribe la siguiente sentencia "*pip install* (nombre de la librería)". Por ejemplo, para instalar la librería *NumPy* se escribe la sentencia "*pip install numpy*" (Figura 1.22) y seguidamente se presiona la tecla "*intro*".

Figura 1.22

Instalación de las librerías de Python.



Nota. Los autores**.**

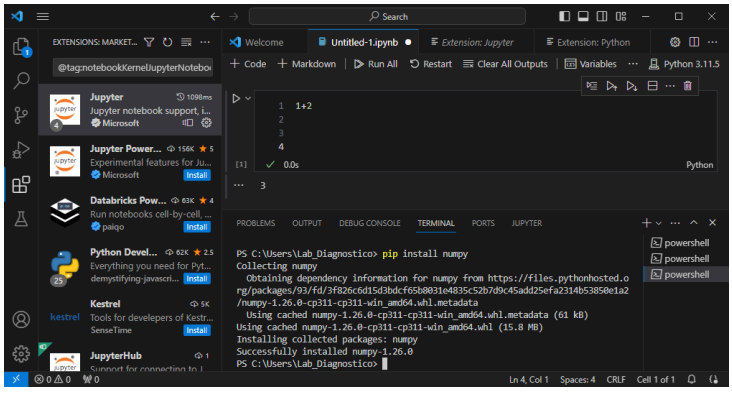
A continuación, se indican las sentencias para la instalación de librerías que se recomienda correr una a la vez:

- *pip install numpy*
- *pip install pandas*
- *pip install openpyxl*
- *pip install scipy*
- *pip install -U scikit-learn*
- *pip install matplotlib*
- *pip install seaborn*

Al finalizar se indica que la instalación se ha realizado satisfactoriamente. En el caso de la librería *NumPy*, sale el mensaje "*Successfully installed numpy*-1.26.0", como se indica en la Figura 1.23.

Figura 1.23

Mensaje final de la instalación de una librería de Python.



Nota. Los autores**.**

1.5 Descripción de las librerías comunes para la estadística de *Python*

Antes de entrar en la materia objeto de la presente obra, se expone una breve descripción de las librerías y las funciones más comunes aplicadas en la estadística descriptiva y análisis de datos, dentro del mantenimiento industrial.

1.5.1 Librería *NumPy*

Proporciona un amplio conjunto de funciones que facilitan la realización de cálculos científicos y matemáticos de valores numérico y datos que pueden estar contenidos en una lista, *tupla*, *DataFrame* o *Series*, incluyendo estadísticas, álgebra lineal, transformadas, generación de números aleatorios y más (Maximov, 2020; Rougier, 2017). La Tabla 1.1 indica algunas de sus funciones.

Tabla 1.1

Principales funciones de NumPy.

N°	Función	Descripción
1	*import numpy as np*	Importa la librería *NumPy*. datos puede ser una lista, *tupla*, *DataFrame* o *Series*.
2	datos = [1, , 3, 4] *np.size*(datos) `Salida: 4`	Calcula el número de elementos de un conjunto de datos.
3	datos = [1, 2, 3, 4] *np.mean*(datos) `Salida: 2.5`	Calcula la media de un conjunto de datos.
4	datos = [1, 2, 3, 4, 5] *np.median*(datos) `Salida: 3.0`	Calcula la mediana de un conjunto de datos.
5	datos = [1, 2, 3, 4] *np.min*(datos) `Salida: 1`	Encuentra el mínimo de un conjunto de datos.
6	datos = [1, 2, 3, 4] *np.max*(datos) `Salida: 4`	Encuentra el máximo de un conjunto de datos.
7	datos = [1, 2, 3, 4] *np.var*(datos) `Salida: 1.25`	Calcula la varianza de un conjunto de datos. Para la varianza poblacional *ddof*=0 (por defecto). Para la varianza muestral *ddof*=1.

N°	Función	Descripción
	np.var(datos, *ddof*=1) `Salida: 1.6666666666666667`	

8	datos = [1, 2, 3, 4]
	
	np.std(datos)
	
	`Salida: 1.118033988749895`
	
	np.std(datos, *ddof*=1)
	
	`Salida: 1.2909944487358056`
		Calcular la desviación estándar de un conjunto de datos.
		Para la desviación estándar poblacional *ddof*=0 (por defecto).
		Para la desviación estándar muestral *ddof*=1.

9	datos = [1, 2, 3, 4]; n = 75 *np.percentile*(datos, n) `Salida: 3.25`	Calcula el percentil "n" de un conjunto de datos ($0 < n < 100$).

10	datos = [1, 2, 3, 4]
	
	Q1, Q2, Q3 = *np.percentile*(datos, [25, 50, 75])
	
	`Salida:`
	`Q1 = 1.75`
	`Q2 = 2.5`
	`Q3 = 3.25`
	
	Q1, Q2, Q3 = *np.quantile*(datos, [0.25, 0.50, 0.75])
	
	`Salida:`
	`Q1 = 1.75`
	`Q2 = 2.5`
	`Q3 = 3.25`
		Calcula los cuartiles (*Q1*, *Q2*, *Q3*) de un conjunto de datos.

11	x = [1, 2, 3, 4, 5] y = [2, 4, 6, 8, 10] np.cov(x, y) [0, 1] `Salida: 5.0`	Calcula la covarianza de dos variables (x, y).

N°	Función	Descripción

12	x = [1, 2, 3, 4, 5]
	y = [2, 4, 6, 8, 10]
	n = 2
	
	b2, b1, b0 = *np.polyfit*(x, y, n)
	
	`Salida:`
	`b0 = 3.972054645195637e-15`
	`b1 = 1.9999999999999996`
	`b2 = 2.046647694443857e-16`
		Calcula la matriz de coeficientes de una regresión polinómica de grado "n" de dos variables (x, y), donde b0 es el intercepto y b1, b2, ... son los coeficientes.

13	x = [1, 2, 3, 4, 5] y = [2, 4, 6, 8, 10] *np.corrcoef*(x, y) [0, 1] `Salida: 0.9999999999999999`	Calcula el coeficiente de correlación r de Pearson de dos variables (x, y).
14	*np.random.seed*(n)	Genera el mismo conjunto de datos aleatorios de semilla n en cada ejecución del programa.

15	a = 3; b = 2
	
	np.random.seed(10)
	np.random.rand(*a*, *b*)
	
	`Salida:`
	`array([[0.77132064, 0.02075195],`
	`[0.63364823, 0.74880388],`
	`[0.49850701, 0.22479665]])`
		Genera un arreglo de "a" filas por "b" columnas de números aleatorios entre 0 y 1.

16	*low* = 0; *high* = 5; n = 4
	
	np.random.seed(10)
	np.random.randint(*low*, *high*, n)
	
	`Salida:`
	`array([1, 4, 0, 1])`
		Genera un arreglo con "n" números enteros aleatorios comprendidos entre "*low*" y "*high*".

17	n = 4 *np.random.seed*(10) *np.random.randn*(n)	Genera un arreglo con "n" números aleatorios con distribución normal estándar.

N°	Función	Descripción
	Salida: array([1.3315865, 0.71527897, -1.54540029, -0.00838385])	
18	mu = 3; sigma = 0.5; n = 4 *np.random.seed*(10) *np.random.normal*(mu, sigma, n) Salida: array([3.66579325, 3.35763949, 2.22729985, 2.99580808])	Genera un arreglo con "n" números aleatorios distribuidos normalmente de media "mu" y desviación estándar "sigma".
19	TMEF = 800; n = 3 *np.random.seed*(10) *np.random.exponential*(TMEF, n) Salida: array([1180.34755644, 16.77623768, 803.32904031])	Genera un arreglo con "n" números aleatorios distribuidos exponencialmente de tiempo entre fallas "TMEF".
20	a = 800; b = 4.2; n = 3 *np.random.seed*(10) *np.random.weibull*(b, n) * a Salida: array([877.62505884, 318.76602783, 800.79137519])	Genera un arreglo con "n" números aleatorios distribuidos de acuerdo con Weibull de parámetro de forma "b" y parámetro de escala "a".
21	vector = [1, 2, 3, 4] *np.random.seed*(10) *np.random.permutation*(vector) Salida: array([3, 1, 4, 2])	Cambia aleatoriamente el orden de los elementos de un vector.
22	datos = [1, 2, 3, 4] *np.sum*(datos) Salida: 10	Calcula la sumatoria de los elementos de un *array*.

N°	Función	Descripción
23	datos = [1, 2, 3, 4] *np.cumsum*(datos) Salida: array([1, 3, 6, 10])	Calcula la suma acumulativa de los elementos de un *array*.
24	datos = [1, 2, 3, 4] *np.prod*(datos) Salida: 24	Calcula la productoria de los elementos de un *array*.
25	datos = [1, 2, 3, 4] *np.cumprod*(datos) Salida: array([1, 2, 6, 24])	Calcula el producto acumulativo de los elementos de un *array*.
26	datos = [1, 4, 9, 16] *np.sqrt*(datos) Salida: array([1., 2., 3., 4.]) *np.sqrt*(4) Salida: 2.0	Calcula la raíz cuadrada de cada elemento de un *array*.
27	datos = [1, 2, 3, 4]; n = 3 *np.power*(datos, n) Salida: array([1, 4, 9, 16], dtype=int32) *np.power*(2, 3) Salida: 8	Calcula la "n" potencia de cada elemento de un *array*.

N°	Función	Descripción
28	datos = [1, 2] *np.exp*(datos) Salida: `array([2.71828183, 7.3890561])` *np.exp*(2) Salida: 7.3890561	Eleva al número de Euler a la potencia indicada en cada elemento de un *array*.
29	datos = [2, 3] *np.log*(datos) Salida: `array([0.69314718, 1.09861229])` *np.log*(2) Salida: 0.69314718	Calcula el logaritmo natural de cada elemento de un *array*.
30	datos = [2, 3] *np.log10*(datos) Salida: `array([0.30103, 0.47712125])` *np.log10*(100) Salida: 2.0	Calcula el logaritmo de base 10 de cada elemento de un *array*.
31	datos = [2, 3] *np.log2*(datos) Salida: `array([1. , 1.5849625])` *np.log2*(8) Salida: 3.0	Calcula el logaritmo de base 2 de cada elemento de un *array*.

N°	Función	Descripción
32	datos = [1, 2]; d = 2 n = *np.exp*(datos) *np.round*(n, d) Salida: array([2.72, 7.39]) *np.round*(3.1416, 3) Salida: 3.141	Redondea un número "n" de acuerdo con el número de decimales "d" (Por defecto $d = 0$).
33	a = ([1.123, 2.124], [3.125, 4.126]) d = 2 *np.around*(a, d) Salida: array([[1.12, 2.12], [3.12, 4.13]])	Redondea la matiz "a" de acuerdo con el número de decimales "d" (Por defecto $d = 0$).
34	datos = [1, 2] n = *np.exp*(datos) *np.ceil*(n) Salida: array([3., 8.]) *np. ceil*(3.1416) Salida: 4.0	Redondea "n" hacia arriba al entero más cercano.
35	datos = [1, 2] n = *np.exp*(datos) *np.floor*(n) Salida: array([2., 7.]) *np.floor*(3.1416) Salida: 3.0	Redondea "n" hacia abajo al entero más cercano.

N°	Función	Descripción

36	n = [1, -2, -3]
	
	np.absolute(n)
	
	`Salida:`
	`Array([1, 2, 3])`
	
	np.absolute(-3)
	
	`Salida: 3`
		Devuelve el valor absoluto de "n".

37	n = 3 *np.prod*(np.arange(1, n + 1)) `Salida: 6`	Calcula el factorial de "n".
38	datos = [2, 3, 4] *np.lcm.reduce*(datos) `Salida: 12`	Calcula el mínimo común múltiplo de un conjunto de datos.
39	datos = [8, 12, 16] *np.gcd.reduce* (datos) `Salida: 4`	Calcula el máximo común divisor de un conjunto de datos.

40	n = [1, 2, *np.inf*, *-np.inf*, *np.nan*]
	
	np.isfinite(n)
	
	`Salida:`
	`array([True, True, False, False, False])`
		Verifica si "n" es un número finito.
		np.inf, es infinito.
		np.nan, es un dato faltante.

41	n = [1, 2, *np.inf*, *-np.inf*, *np.nan*]
	
	np.isnan(n)
	
	`Salida:`
	`array([False, False, False, False, True])`
		Verifica si "n" es *NaN* (no es un número o dato faltante).
		np.inf, es infinito.
		np.nan, es un dato faltante.

N°	Función	Descripción
42	*np.e* `Salida: 2.718281828459045`	Número de Euler.
43	*np.pi* `Salida: 3.141592653589793`	Constante pi (3,14159...).

44	n = [30, 45, 60, 180]
	
	np.radians(n)
	
	`Salida:`
	`array([0.52359878, 0.78539816, 1.04719755, 3.14159265])`
	
	np.radians(180)
	
	`Salida: 3.141592653589793`
		Convierte ángulos de grados a radianes.

45	n = [*np.pi*/6, *np.pi*/4, *np.pi*/3]
	
	np.degrees(n)
	
	`Salida:`
	`array([30., 45., 60.])`
	
	np.degrees(*np.pi*)
	
	`Salida: 180.0`
		Convierte ángulos de radianes a grados.

46	n = [*np.pi*/6, *np.pi*/4, *np.pi*/3]
	
	np.sin(n)
	
	`Salida:`
	`array([0.5, 0.70710678, 0.8660254])`
	
	np.cos(n)
	
	`Salida:`
	`array([0.8660254, 0.70710678, 0.5])`
		Calcula las funciones trigonométricas seno, coseno y tangente de “n” en radianes, respectivamente.

N°	Función	Descripción
	np.tan(n) Salida: array([0.57735027, 1., 1.73205081]) [*np.sin*(*np.pi*/6), *np.cos*(*np.pi*/3), *np.tan*(*np.pi*/4)] Salida: [0.4999999999999994, 0.5000000000000001, 0.9999999999999999] *np.tan*(*np.radians*(90)) Salida: 1.633123935319537e+16	
47	*start* = 0; *stop* = 5; h = 1 *np.arange*(*start*, *stop*, h) Salida: array([0, 1, 2, 3, 4]) *np.arange*(5) Salida: array([0, 1, 2, 3, 4])	Crea un arreglo de valores espaciados a "h".
48	*start* = 0; *stop* = 5; n = 6 *np.linspace*(start, stop, n) Salida: array([0., 1., 2., 3., 4., 5.])	Crea un arreglo con "n" valores equidistantes en un rango.
49	datos = [1, 2, 3, 4, 5, 6] a = 2; b = 3 *np.reshape*(datos, [a, b]) Salida:	Cambia la forma del arreglo "datos" de (a*b) elementos, a otro de dimensiones "a" filas * "b" columnas.

N°	Función	Descripción
	`array([[1, 2, 3],` `[4, 5, 6]])`	

50	datos = *np.array*([[1, 2, 3],
	[4, 5, 6]])
	datos2 = *np.array*([[7, 8, 9]])
	
	np.concatenate((datos, datos2), axis=0)
	
	`Salida:`
	`array([[1, 2, 3],`
	`[4, 5, 6],`
	`[7, 8, 9]])`
		Coloca las filas del arreglo "datos2" a continuación de la última fila del arreglo "datos" (concatenación vertical).

51	datos = *np.array*([[1, 2, 3],
	[4, 5, 6]])
	datos2 = *np.array*([[7, 8],
	[9, 10]])
	
	np.concatenate((datos, datos2), axis=1)
	
	`Salida:`
	`array([[1, 2, 3, 7, 8],`
	`[4, 5, 6, 9, 10]])`
		Coloca las columnas del arreglo "datos2" a continuación de la última columna del arreglo "datos" (concatenación horizontal).

52	datos = [1, 2, 3, 4]; n = 2
	
	np.delete(datos, n)
	
	`Salida:`
	`Array([1, 2, 4])`
	
	np.delete(datos, -1)
	
	`Salida:`
	`array([1, 2, 3])`
	
	np.delete(datos, [-1, -2])
	
	`Salida:`
	`array([1, 4])`
		Elimina el elemento de *index* "n" del arreglo "datos". Si "n" es negativo, "n" se cuenta desde el último elemento.

N°	Función	Descripción

53	datos = [1, 2, 3, 4, 5]; n = 2
	
	datos [:n]
	
	`Salida: [1, 2]`
	
	datos [:-2]
	
	`Salida: [1, 2, 3]`
		Extrae los "n" primeros elementos del arreglo "datos". Si "n" es negativo, se eliminan los últimos "n" elemento. Si $n = -1$, se elimina únicamente el último elemento.

54	datos = [1, 2, 3, 4, 5] datos.*pop*() `Salida: 5`	Es una función directa de *Python* (no es de *NunPy*) para seleccionar el último elemento de la lista "datos".

55	datos = *np.array*([1, 2, 3, 4, 5])
	m = 3
	
	np.delete(datos, *np.where*(datos == m))
	
	`Salida:`
	`array([1, 2, 4, 5])`
		Elimina el elemento "m" del arreglo "datos". Si "m" es un *str* (cadena de caracteres) debe estar entre comillas.

56	x = *np.array*([1, 2, 3])
	y = *np.array*([4, 5, 6])
	
	np.dot(x, y)
	
	`Salida: 32`
	
	x = *np.array*([[1, 2, 3],
	[4, 5, 6]])
	y = *np.array*([[7, 8],
	[9, 10],
	[11, 12]])
	
	np.dot(x, y)
	
	`Salida:`
	`array([[ 58,  64],`
	`[139, 154]])`
		Producto punto entre los arreglos "x" y "y".

N°	Función	Descripción

57	x = *np.array*([1, 2, 3])
	y = *np.array*([4, 5, 6])
	
	np.matmul(x, y)
	
	`Salida: 32`
	
	x = *np.array*([[1, 2, 3],
	[4, 5, 6]])
	y = *np.array*([[7, 8],
	[9, 10],
	[11, 12]])
	
	np.matmul(x, y)
	
	`Salida:`
	`array([[ 58,  64],`
	`[139, 154]])`
		Igual que *np.dot*(). Es especialmente útil cuando se estás trabajando con código más legible y específico para la multiplicación de matrices.

58	x = *np.array*([[1, 2], [3, 4]]) *np.linalg.norm*(x) `Salida: 5.477225575051661`	Calcula la norma (longitud) del arreglo "x".
59	x = *np.array*([[1, 2], [3, 4]]) *np.linalg.det*(x) `Salida: -2.0000000000000004`	Calcula el determinante de la matriz "x".

60	x = *np.array*([[1, 2], [3, 4]])
	
	np.linalg.inv(x)
	
	`Salida:`
	`array([[-2. ,  1. ],`
	`[ 1.5, -0.5]])`
		Calcula la matriz inversa de "x".

Nota. Los autores.

1.5.2 Librería *Pandas*

Es utilizada para la manipulación y análisis de datos tabulares y series temporales. Ofrece estructuras de datos flexibles, como *DataFrames* o *Series*, que permiten cargar, limpiar, transformar y analizar datos de manera eficiente. Pandas facilita la indexación, filtrado y agregación de datos (Pandas, 2023). La Tabla 1.2 indica varias de sus funciones.

Tabla 1.2

Principales funciones de Pandas.

N°	Función	Descripción
1	*import pandas as pd*	Importa la librería *Pandas*. datos puede ser un objeto *DataFrame* o un objeto *Series*.
2	datos = *pd.read_csv*("nombre_archivo.csv")	Crea el objeto "datos" y le asigna la información importada desde un archivo *csv* o *txt*.
3	datos = *pd.read_excel*(("nombre_archivo.xlsx")	Crea el objeto "datos" y le asigna la información importada desde un archivo *xlsx*.
4	datos = *pd.read_sql*(("nombre_archivo.sql")	Crea el objeto "datos" y le asigna la información importada desde un archivo *sql*.

N°	Función	Descripción
5	datos = *pd.read_json*(("nombre_archivo.json")	Crea el objeto "datos" y le asigna la información importada desde un archivo *json*.
6	datos.*to_csv*("nombre_archivo.csv")	Guarda un *DataFrame* en un archivo *csv*.
7	datos.*to_excel*("nombre_archivo.xlsx", *sheet_name*="nombre_hoja", *index=False*)	Guarda un *DataFrame* en un archivo *xlsx*.
8	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) Salida: A B 0 1 11 1 2 12 2 3 13	Crea el objeto "datos" y le asigna un *DataFrame* creado a partir de un diccionario, con las columnas "A" con los registros (1, 2, 3), y "B" con los registros (11, 12, 13).
9	datos = *pd.Series*([1, 2, 3]) Salida: 0 1 1 2 2 3 dtype: int64	Crea el objeto "datos" y le asigna un conjunto de datos tipo *Series* creado a partir de una lista, con los registros (1, 2, 3).
10	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) columna = datos.*loc*[:,'B'] Salida: 0 11 1 12 2 13 Name: B, dtype: int64	Crea el objeto "columna" y se le asigna los valores de la columna "B" del *DataFrame* "datos" con todos sus registros.
11	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) n = 1	Crea el objeto "columna" y se le asigna los valores de la columna de índice "n" del *DataFrame "datos"*.

N°	Función	Descripción
	columna = datos.*iloc*[:,n] Salida: 0 11 1 12 2 13 Name: B, dtype: int64	
11	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) n = 0 fila = datos.*iloc*[n] Salida: A 1 B 11 Name: 0, dtype: int64	Crea el objeto "fila" y se le asigna los valores de la fila de índice "n" del *DataFrame "datos"*.
13	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) n = 0 datos = datos.*drop*(n) Salida: A B 1 2 12 2 3 13 datos.*drop*(n, *inplace=True*) Salida: A B 1 2 12 2 3 13	Elimina la fila de *index* "n" del *DataFrame* "datos".
14	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) datos = datos.*drop*(datos.*index*[-1]) Salida: A B 0 1 11 1 2 12	Elimina la última fila del *DataFrame* "datos".

N°	Función	Descripción
	datos.*drop*(datos.*index*[-1], *inplace*=*True*) Salida: **A B** 0 1 11 1 2 12	
15	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) datos = datos.*sort_index*() datos.*sort_index*(*inplace*=*True*) Salida: **A B** 0 1 11 1 2 12 2 3 13 datos = datos.*sort_index*(*ascending*=*False*) datos.*sort_index*(*ascending*=*False*, *inplace*=*True*) Salida: **A B** 2 3 13 1 2 12 0 1 11	Reordena el *index* ascendentemente. Si se añade el argumento *ascending=False*, el *index* se reordena descendentemente.
16	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) datos = datos.*drop*('A', *axis*=1) datos.*drop*('A', *axis*=1, *inplace*=*True*) Salida: **B** 0 11 1 12 2 13	Elimina la columna de encabezado "A" del *DataFrame* "datos".

N°	Función	Descripción

17	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]})
	
	datos = datos.*rename*(*columns*={'A': 'C'})
	
	datos.*rename*(*columns*={'A': 'C'}, *inplace*=*True*)
	
	Salida:
	`  C  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
		Renombrar la columna "A" a "B" del *DataFrame* "datos".

18	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]})
	n = 0
	
	datos = datos.*rename*(*index*={n: 'nombre'})
	
	datos.*rename*(*index*={n: 'nombre'}, *inplace*=*True*)
	
	Salida:
	`  A  B`
	`nombre 1 11`
	`1 2 12`
	`2 3 13`
		Renombrar la fila en el índice "n" a "nombre" del *DataFrame* "datos".

19	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, np.nan, 13]})
	`  A  B`
	`0 1 11.0`
	`1 2 NaN`
	`2 3 13.0`
	
	datos = datos.*dropna*()
	
	Salida:
	`  A  B`
	`0 1 11.0`
	`2 3 13.0`
		Elimina las filas que contienen al menos un valor *NaN* (valor faltante) del *DataFrame* "datos".

N°	Función	Descripción

20	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, np.nan, 13]})
	`    A      B`
	`0   1   11.0`
	`1   2    NaN`
	`2   3   13.0`
	
	datos = datos.*dropna*(*axis*=1)
	
	`Salida:`
	`    A`
	`0   1`
	`1   2`
	`2   3`
		Elimina las columnas que contienen al menos un valor *NaN* (valor faltante) del *DataFrame* "datos".

21	datos = *pd.DataFrame*({'A':[np.nan, 2, 3], 'B':[11, np.nan, 13]})
	`     A      B`
	`0  NaN   11.0`
	`1    2    NaN`
	`2    3   13.0`
	
	datos = datos.*dropna*(*subset*=['A'])
	
	`Salida:`
	`      A      B`
	`1  2.0    NaN`
	`2  3.0  13.0`
		Elimina las filas que contienen al menos un valor *NaN* (valor faltante) sólo de la columna "A" del *DataFrame* "datos".

22	datos = *pd.DataFrame*({'A':[np.nan, 2, 3], 'B':[11, np.nan, 13]})
	`     A      B`
	`0  NaN   11.0`
	`1    2    NaN`
	`2    3   13.0`
	n = 22
	
	datos['A'].*fillna*(n, *inplace*=*True*)
	
	`Salida:`
	`       A      B`
	`0  22.0  11.0`
	`1   2.0   NaN`
	`2   3.0  13.0`
		Rellena los valores faltantes en la columna "A" del *DataFrame* "datos", con el valor "n".

N°	Función	Descripción

23	datos = *pd.DataFrame*({'A':[np.nan, 2, 3], 'B':[11, np.nan, 13]})
	`  A  B`
	`0 NaN 11.0`
	`1  2  NaN`
	`2  3  13.0`
	
	mediana = datos['A'].*median*()
	datos['A'].*fillna*(mediana, *inplace*=*True*)
	
	`Salida:`
	`  A  B`
	`0 2.5 11.0`
	`1 2.0 NaN`
	`2 3.0 13.0`
		Rellenar los valores faltantes en la columna "A" del *DataFrame* "datos", con la mediana de esa columna.

24	datos = *pd.DataFrame*({'A':[np.nan, 2, 3], 'B':[11, np.nan, 13]})
	`  A  B`
	`0 NaN 11.0`
	`1  2  NaN`
	`2  3  13.0`
	
	datos['B'].*fillna*(*method*='ffill', *inplace*=*True*)
	
	`Salida:`
	`  A  B`
	`0 NaN 11.0`
	`1 2.0 11.0`
	`2 3.0 13.0`
		Rellenar los valores faltantes en la columna "B" del *DataFrame* "datos", con el valor anterior en la misma columna.

25	datos = *pd.DataFrame*({'A':[1, 2, 3],
	'B':[11, 12, 13]})
	`  A  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
	
	datos2 = *pd.DataFrame*({'C':[21], 'D':[31]})
	`  C  D`
	`0 21 31`
		Agregar filas adicionales (concatena verticalmente "datos" con "datos2").

N°	Función	Descripción
	datos3 = *pd.DataFrame*() datos3['E'] = *pd.concat*([datos['A'], datos2['C']], *axis*=0) Salida: **E** **0** 1 **1** 2 **2** 3 **0** 21	
26	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'C':[21], 'D':[31]}) C D 0 21 31 datos3 = *pd.DataFrame*() datos3['E'] = *pd.concat*([datos['A'], datos2['C']], *ignore_index=True*) Salida: **E** **0** 1 **1** 2 **2** 3 **3** 21	Agregar filas adicionales (concatena verticalmente "datos" con "datos2") sin tener en cuenta los índices originales.
27	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'C':[21], 'D':[31]})	Agregar columnas adicionales (concatena horizontalmente "datos" con "datos2").

N°	Función	Descripción
	C D 0 21 31 datos3 = *pd.DataFrame*() datos3['E'] = *pd.concat*([datos['A'], datos2['C']], *axis*=1) Salida: A B C D 0 1 11 21.0 31.0 1 2 12 NaN NaN 2 3 13 NaN NaN	
28	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'A':[2, 3, 4], 'B':[51, 52, 53]}) A B 0 2 51 1 3 52 2 4 53 datos = *pd.merge*(datos, datos2, *on*='A') Salida: A B_x B_y 0 2 12 51 1 3 13 52	Obtiene las filas de "datos" y "datos2" que tengan coincidencias con la columna "A".
29	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'A':[2, 3, 4], 'B':[51, 52, 53]})	Obtiene las filas de "datos" y "datos2" que tengan coincidencias con la columna "A". Igual que: datos = *pd.merge*(datos, datos2, *on*='A')

N°	Función	Descripción
	A B 0 2 51 1 3 52 2 4 53 datos = *pd.merge*(datos, datos2, *on*='A', *how*='*inner*') Salida: **A B_x B_y** 0 2 12 51 1 3 13 52	
30	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'A':[2, 3, 4], 'B':[51, 52, 53]}) A B 0 2 51 1 3 52 2 4 53 datos = *pd.merge*(datos, datos2, *on*='A', *how*='*outer*') Salida: **A B_x B_y** 0 1 11.0 NaN 1 2 12.0 51.0 2 3 13.0 52.0 3 4 NaN 53.0	Obtiene todas las filas de "datos" y "datos2", con valores coincidentes y NaN donde no hay coincidencias.
31	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'A':[2, 3, 4], 'B':[51, 52, 53]})	Obtiene las filas de "datos" y las filas de "datos2" que tengan coincidencias con la columna "A".

N°	Función	Descripción
	A B 0 2 51 1 3 52 2 4 53 datos = *pd.merge*(datos, datos2, *on*='A', *how*='*left*') Salida: **A B_x B_y** 0 1 11 NaN 1 2 12 51.0 2 3 13 52.0	
32	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos2 = *pd.DataFrame*({'A':[2, 3, 4], 'B':[51, 52, 53]}) A B 0 2 51 1 3 52 2 4 53 datos = *pd.merge*(datos, datos2, *on*='A', *how*='*right*') Salida: **A B_x B_y** 0 2 12.0 51 1 3 13.0 52 2 4 NaN 53	Obtiene las filas de "datos2" y las filas de "datos" que tengan coincidencias con la columna "A".
33	datos = *pd.DataFrame*({'A':['María', 'Pedro', 'Ana'], 'B':[11, 12, 13]}) A B 0 María 11 1 Pedro 12 2 Ana 13	Devuelve la fila en la que la columna "A" contiene la palabra "a", sin importar mayúsculas/minúsculas.

N°	Función	Descripción
	datos[datos['A'].*str.contains*('a' , *case*=*False*)] Salida: **A B** 0 María 11 2 Ana 13	
34	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 2, 13]}) A B 0 1 11 1 2 2 2 3 13 a = 2; b = 44 datos = datos.*replace*(a, b) Salida: **A B** 0 1 11 1 44 44 2 3 13	Reemplaza "a" con "b" en todo el *DataFrame* "datos".
35	datos = *pd.DataFrame*({'A':['María', 'Pedro', 'Ana'], 'B':[11, 12, 13]}) A B 0 María 11 1 Pedro 12 2 Ana 13 a = 'Pedro'; b = 'Juan' datos['A'] = datos['A'].*replace*(a, b) datos['B'] = datos['B'].*replace*(12, 22) Salida: **A B** 0 María 11 1 Juan 22 2 Ana 13	Reemplazar "a" con "b" en la columna "A" del *DataFrame* "datos".

N°	Función	Descripción
36	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 3 *np.random.seed*(10) datos2 = datos.*sample*(n) Salida: A B 0 1 11 2 3 13 1 2 12	Crea la muestra "datos2" de "n" filas aleatorias y el mismo número de columnas del *DataFrame* "datos".
37	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 3 *np.random.seed*(10) datos2 = datos.*sample*(n, *replace*=*True*) Salida: A B 1 2 12 1 2 12 0 1 11	Crea la muestra "datos2" de "n" filas aleatorias con reemplazamiento (las filas pueden repetirse) y con el mismo número de columnas del *DataFrame* "datos".
38	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 2 *np.random.seed*(0) datos2 = datos.*sample*(n, *axis*=1)	Crea la muestra "datos2" de "n" columnas aleatorias y el mismo número de filas del *DataFrame* "datos".

N°	Función	Descripción

	Salida:
	`  B  A`
	`0 11  1`
	`1 12  2`
	`2 13  3`	

	datos = *pd.DataFrame*({'A':[1, 2, 3],
	'B':[11, 12, 13]})
	`  A  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
	n = 5
	
	np.random.seed(10)
	datos2 = datos.*sample*(n, *axis*=1, *replace*=*True*)
	
	Salida:
	`  B  B  A  B  A`
	`0 11 11  1 11  1`
	`1 12 12  2 12  2`
	`2 13 13  3 13  3`
		Crea la muestra "datos2" de "n" columnas aleatorias de "n" filas aleatorias con reemplazamiento (las filas pueden repetirse) y el mismo número de filas del *DataFrame* "datos".

39	datos = *pd.DataFrame*({'A':[1, 2, 3],
	'B':[11, 12, 13]})
	`  A  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
	
	datos.*mean*()
	
	Salida:
	`A    2.0`
	`B   12.0`
	`dtype: float64`
		Calcula la media de un conjunto de datos.

40	datos = *pd.DataFrame*({'A':[1, 2, 2],
	'B':[11, 12, 12]})
	`  A  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
		Genera una matriz con las modas de un conjunto de datos.

N°	Función	Descripción
	datos.*mode*() Salida: A B 0 2 12	
41	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*median*() Salida: A 2.0 B 12.0 dtype: float64	Calcula la mediana de un conjunto de datos.
42	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*min*() Salida: A 1 B 11 dtype: int64	Encuentra el mínimo de un conjunto de datos.
43	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13	Encuentra el máximo de un conjunto de datos.

N°	Función	Descripción
	datos.*max*() Salida: A 3 B 13 dtype: int64	
44	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*var*() Salida: A 1.0 B 1.0 dtype: float64	Calcula la varianza muestral de un conjunto de datos.
45	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*std*(*ddof*=1) Salida: A 1.0 B 1.0 dtype: float64	Si *ddof*=1 (por defecto) calcula la desviación estándar muestral de un conjunto de datos. Si *ddof*=0, se obtiene la desviación estándar poblacional.
46	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 0.28 datos.quantile(n)	Calcula el cuantil "n" de un conjunto de datos ($0 < n < 1$).

N°	Función	Descripción
	Salida: A 1.56 B 11.56 Name: 0.28, dtype: float64	
47	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.quantile([0.25, 0.5, 0.75]) Salida: A B 0.25 1.5 11.5 0.50 2.0 12.0 0.75 2.5 12.5	Calcula los cuartiles 1, 2 y 3 de un conjunto de datos (*Q1*, *Q2* y *Q3*).
48	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*sum*() Salida: A 6 B 36 dtype: int64	Calcula la sumatoria de un conjunto de datos.
49	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*cumsum*()	Calcula la suma acumulativa de los elementos de un conjunto de datos.

N°	Función	Descripción
	Salida: A B 0 1 11 1 3 23 2 6 36	
50	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*prod*() Salida: A 6 B 1716 dtype: int64	Calcula la productoria de un conjunto de datos.
51	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos.*cumprod*() Salida: A B 0 1 11 1 2 132 2 6 1716	Calcula el producto acumulativo de los elementos de un conjunto de datos.
52	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 2 datos.*pow*(n)	Calcula la "n" potencia de cada elemento de un conjunto de datos.

N°	Función	Descripción
	Salida: A B 0 1 121 1 4 144 2 9 169	
53	datos = *pd.DataFrame*({'A':[1, 2, 2], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 2 13 datos2 = datos.groupby('A')['B'].sum() Salida: A 1 11 2 25 Name: B, dtype: int64	Crea grupos de los valores de la columna "B" de acuerdo con la columna "A" y aplica una operación (en este caso la suma).
54	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 2 datos.*head*(n) Salida: A B 0 1 11 1 2 12	Muestra los "n" primeras filas del *DataFrame* "datos". Por defecto $n = 5$.
55	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 n = 2 datos.*tail*(n)	Muestra los "n" últimas filas del *DataFrame* "datos". Por defecto $n = 5$.

N°	Función	Descripción

	Salida:
	`  A  B`
	`1 2 12`
	`2 3 13`	

56	`datos = pd.DataFrame({'A':[1, 2, 3], 'B':[11, 12, 13]})`
	`  A  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
	
	`datos.describe()`
	
	Salida:
	`  A  B`
	`count 3.0 3.0`
	`mean 2.0 12.0`
	`std 1.0 1.0`
	`min 1.0 11.0`
	`25% 1.5 11.5`
	`50% 2.0 12.0`
	`75% 2.5 12.5`
	`max 3.0 13.0`
		Determina al mismo tiempo: *count*, *mean*, *std*, *min*, *Q1*, *Q2*, *Q3*, *max* y *type*, de un conjunto de datos.

57	`datos = pd.DataFrame({'A':[1, 2, 3], 'B':[11, 12, 13]})`
	`  A  B`
	`0 1 11`
	`1 2 12`
	`2 3 13`
	
	`datos.info()`
	
	Salida:
	<class 'pandas.core.frame.DataFrame'>
	RangeIndex: 3 entries, 0 to 2
	Data columns (total 2 columns):
	`# Column Non-Null Count Dtype`
	`--- ------ -------------- -----`
	`0 A 3 non-null int64`
	`1 B 3 non-null int64`
		Proporciona la información técnica sobre el *DataFrame* "datos" como el número de columnas, los encabezados, número de datos no nulos por columna, tipo de dato por columnas, entre otros.

N°	Función	Descripción
	dtypes: int64(2) memory usage: 180.0 bytes	
58	datos = *pd.DataFrame*({'A':[1, 2, 3], 'B':[11, 12, 13]}) A B 0 1 11 1 2 12 2 3 13 datos['A'].*tolist*() Salida: [1, 2, 3]	Convierte los elementos de la columna 'A' del *DataFrame* "datos" en una lista.

Nota. Los autores**.**

1.5.3 Librería *Openpyxl*

Se utiliza para trabajar con archivos de Excel (.xlsx). Permite la creación, lectura y manipulación de hojas de cálculo de forma programática, lo que es especialmente útil en aplicaciones de procesamiento de datos, acceder a celdas, dar formato a hojas de cálculo, crear gráficos y realizar muchas otras operaciones relacionadas con Excel (Charlie, 2023). La Tabla 1.3 indica varias de sus funciones habituales.

Tabla 1.3

Principales funciones de Openpyxl.

N°	Función	Descripción
1	*import openpyxl as xl*	Importa la librería *Openpyxl*.
2	*xl.load_workbook*()	Permite cargar un archivo de Excel existente en *Python*.
3	*xl.Workbook*()	Crea un nuevo archivo de Excel en blanco.
4	Sheetnames	Propiedades que devuelve una lista de nombres de hojas de trabajo en el archivo de Excel.
5	get_sheet_by_name()	Permite obtener una hoja de trabajo por su nombre.
6	*active*	Propiedades que devuelve la hoja de trabajo activa actual.
7	cell()	Permite acceder a una celda específica por su coordenada (por ejemplo, "A1").
8	value	Propiedad que permite obtener o establecer el valor de una celda.
9	row	Propiedades que permiten obtener la fila de una celda.
10	column	Propiedades que permiten obtener la columna de una celda.
11	save()	Permite guardar cambios en un archivo de Excel.

N°	Función	Descripción
12	save_as()	Permite guardar una copia del archivo con un nombre diferente.
13	close()	Cierra el archivo de Excel.

Nota. Los autores**.**

1.5.4 Librería *SciPy*

Se construye sobre *NumPy* y proporciona un conjunto de funciones y algoritmos avanzados para realizar análisis numéricos y resolver problemas matemáticos y científicos complejos. Incluye módulos para la optimización, interpolación, transformadas de Fourier, procesamiento de señales, álgebra lineal, estadísticas, integración numérica y más (SciPy, 2023a). La Tabla 1.4 indica una breve descripción de algunas de sus sublibrerías.

Tabla 1.4

Principales sublibrerías de SciPy.

N°	Sublibrería	Descripción
1	*scipy.special*	Proporciona funciones matemáticas especiales, como la función gamma y las funciones de Bessel.

N°	Sublibrería	Descripción
2	*scipy.integrate*	Ofrece herramientas para realizar integración numérica y resolver ecuaciones diferenciales ordinarias (ODEs) y ecuaciones diferenciales parciales (PDEs).
3	*scipy.optimize*	Proporciona un conjunto de algoritmos de optimización numérica no lineal, lineal, y global para encontrar máximos o mínimos.
4	*scipy.interpolate*	Proporciona funciones para realizar interpolación de datos en 1D, 2D y 3D.
5	*scipy.fft*	Permite realizar transformadas de Fourier en señales y datos.
6	*scipy.signal*	Proporciona una amplia variedad de herramientas para el procesamiento de señales, incluyendo filtrado, análisis espectral, convolución y correlación.
7	*scipy.linalg*	Proporciona funciones para realizar operaciones de álgebra lineal, como descomposiciones de matrices, solución de sistemas de ecuaciones lineales y cálculo de valores y vectores propios.
8	*scipy.sparse*	Ofrece estructuras de datos y operaciones en matrices dispersas (*sparse*) para ahorrar memoria y tiempo de cálculo.
9	*scipy.spatial*	Contiene algoritmos y estructuras de datos para datos espaciales, como búsqueda de vecinos cercanos y cálculo de distancias.

N°	Sublibrería	Descripción
10	*scipy.stats*	Ofrece una amplia gama de distribuciones de probabilidad, pruebas estadísticas, generadores de números aleatorios y herramientas para realizar análisis estadísticos.
11	*scipy.ndimage*	Permite realizar operaciones de procesamiento de imágenes, como filtrado, convolución, morfología matemática y análisis de etiquetas.
12	*scipy.io*	Proporciona funciones para leer y escribir datos desde/hacia archivos en varios formatos, como MATLAB, NetCDF, entre otras.
13	*scipy.cluster*	Contiene funciones para el *clustering* y agrupamiento de datos, como el *clustering* jerárquico y *K-Means*.
14	*scipy.constants*	Proporciona constantes matemáticas y físicas, como π (pi) y la velocidad de la luz en el vacío (c).
15	*scipy.odr*	Ofrece funciones para realizar regresión ortogonal (*Orthogonal Distance Regression, ODR*).
16	*scipy.weave*	Permite la integración de código C/C++ en *Python* para mejorar el rendimiento de cálculos numéricos.

Nota. Los autores**.**

La sublibrería *scipy.stats*, proporciona una amplia gama de distribuciones estadísticas, como (SciPy, 2023b):

- *uniform* (uniforme);

- *randint* (uniforme entero entre);
- *binom* (binomial);
- *poisson* (Poisson);
- *norm* (normal);
- *chi2* (chi-cuadrado);
- *t* (t Student);
- *expon* (exponencial);
- *weibull_min* (Weibull);
- *gamma* (gamma);
- entre otras.

Con estas distribuciones se pueden realizar una serie de operaciones como las indicadas a continuación:

- función de masa de probabilidades (*pmf*) para las variables discretas;
- función de densidad de probabilidad (*pdf*) para las variables continuas;
- función de probabilidades acumulada (*cdf*);
- percentil de una probabilidad acumulada (*ppf*);
- valores aleatorios de una distribución (*rvs*);
- parámetros de una distribución (*fit*);

- media, varianza, simetría y curtosis (mvsk);
- límites de confianza (*interval*);
- entre otras.

La sublibrería *scipy.stats* también contiene funciones para hacer pruebas de normalidad de las muestras y de hipótesis, así como también para encontrar los parámetros de regresiones lineales y no lineales, entre otros. La Tabla 1.5 indica varias de sus principales funciones.

Tabla 1.5

Principales funciones de scipy.stats.

N°	Función	Descripción
1	*from scipy.stats import **	Importa las funciones de la sublibrería *stats* de la librería *SciPy*.

2	x = 5; *loc* = 2; *scale* = 8
	uniform.pdf(x, *loc*, *scale*)
	`Salida: 0.125`
	x = 5; *loc* = 2; *scale* = 8
	uniform.cdf(x, *loc*, *scale*)
	`Salida: 0.375`
	f = 0.375; *loc* = 2; *scale* = 8
	uniform.ppf(f, *loc*, *scale*)
	`Salida: 5.0`
		x son los valores que va a tomar la variable aleatoria.
		loc es el límite inferior. Por defecto $loc = 0$.
		scale es el rango del intervalo (diferencia entre el límite superior e inferior). Por defecto $scale = 1$.
		f es una probabilidad.
		size es el número de elementos.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

	loc = 2; *scale* = 8; m = 3
	np.random.seed(10)
	uniform.rvs(*loc*, *scale*, *size*=m)
	
	`Salida: array([8.17056515, 2.16601559, 7.06918588])`
	
	loc = 2; *scale* = 8
	uniform.stats(*loc*, *scale*, moments= 'mvsk')
	
	`Salida: (6.0, 5.33, 0.0, -1.2)`
	
	datos = [7, 5, 6]
	uniform.fit(datos)
	
	`Salida: (5.0, 2.0)`
	
	confianza = 0.95; *loc* = 2; *scale* = 8
	uniform.interval(confianza, *loc*, *scale*)
	
	`Salida: (2.2, 9.8)`	

4	k = 3; n = 8; p = 0.20
	binom.pmf(k, n, p)
	
	`Salida: 0.14680063999999998`
	
	k = 3; n = 8; p = 0.20
	binom.cdf(k, n, p)
	
	`Salida: 0.9437184`
	
	f = `0.9437184`; n= 8; p = 0.20
	binom.ppf(f, n, p)
	
	`Salida: 3.0`
	
	n = 8; p = 0.20; m = 3
	np.random.seed(10)
	binom.rvs(n, p, *size*=m)
		$f(k,n,p) = \binom{n}{k} p^k q^{(n-k)}$
		
		p es la probabilidad de éxito en cada ensayo.
		q es la probabilidad de fracaso en cada ensayo, $q = 1 - p$.
		n es el número de ensayos.
		k es el número de éxitos que pueden ocurrir entre los n ensayos.
		f es una probabilidad.
		size es el número de elementos.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

	`Salida: array([2, 0, 2], dtype=int64)`
	
	n = 8; p = 0.20
	binom.stats(n, p, moments='mvsk')
	
	`Salida: (1.6, 1.28, 0.53, 0.031)`
	
	confianza = 0.95; n = 8; p = 0.20
	binom.interval(confianza, n, p)
	
	`Salida: (0.0, 4.0)`	

5	k = 5; mu = 2
	poisson.pmf(k, mu)
	
	`Salida: 0.03608940886309672`
	
	k = 5; mu = 2
	poisson.cdf(k, mu)
	
	`Salida: 0.9834363915193856`
	
	k = 0.9834363915193856; mu = 2
	poisson.ppf(f, mu)
	
	`Salida: 4.0`
	
	mu = 2; m = 3
	np.random.seed(10)
	poisson.rvs(mu, *size*=m)
	
	`Salida: array([1, 3, 2], dtype=int64)`
	
	mu = 2
	poisson.stats(mu, moments='mvsk')
	
	`Salida: (2.0, 2.0, 0.7071067811865476, 0.5)`
		$f(k,\mu) = \frac{\mu^k}{k!}e^{-\mu}$
		
		k es el número de ocurrencias.
		μ *o* mu es la media y la varianza.
		f es una probabilidad.
		size es el número de elementos.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción
	confianza = 0.95; mu = 2 *poisson.interval*(confianza, mu) `Salida: (0.0, 5.0)`	

6	x = 5; *loc* = 4; *scale* = 0.5
	norm.pdf(x, *loc*, *scale*)
	
	`Salida: 0.10798193302637613`
	
	x = 5; *loc* = 4; *scale* = 0.5
	norm.cdf(x, *loc*, *scale*)
	
	`Salida: 0.9772498680518208`
	
	f = 0. 9772498680518208; *loc* = 4; *scale* = 0.5
	norm.ppf(f, *loc*, *scale*)
	
	`Salida: 5.0`
	
	loc = 4; *scale* = 0.5; m = 3
	np.random.seed(10)
	norm.rvs(*loc*, *scale*, *size*=m)
	
	`Salida: array([4.66579325, 4.35763949, 3.22729985])`
	
	loc = 4; *scale* = 0.5
	norm.stats(*loc*, *scale*, moments='mvsk')
	
	`Salida: (4.0, 0.25, 0.0, 0.0)`
	
	datos = [4.66579, 4.35764, 3.22720]
	norm.fit(datos)
	
	`Salida: (4.0835433, 0.61845616)`
	
	confianza = 0.95; *loc* = 4; *scale* = 0.5
	norm.interval(confianza, *loc*, *scale*)
	
	`Salida: (3.020018007729973, 4.979981992270027)`
		$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$
		
		x son los valores que va a tomar la variable aleatoria.
		$\mu \rightarrow loc$ es la media poblacional. Por defecto $loc = 0$.
		$\sigma \rightarrow scale$ es la desviación estándar poblacional. Por defecto $scale = 1$.
		f es una probabilidad.
		size es el número de elementos.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

7	x = 5; df = 14; loc = 4; $scale$ = 0.5
	t.pdf(x, df, loc, $scale$)
	
	`Salida: 0.11901668155857803`
	
	x = 5; df = 14; loc = 4; $scale$ = 0.5
	t.cdf(x, df, loc, $scale$)
	
	`Salida: 0.967356023555444`
	
	f = 0.967356023555444; df = 14; loc = 4; $scale$ = 0.5
	t.ppf(f, df, loc, $scale$)
	
	`Salida: 4.999999999999867`
	
	df = 14; loc = 4; $scale$ = 0.5; m = 3
	np.random.seed(10)
	t.rvs(df, loc, $scale$, *size*=m)
	
	`Salida: array([4.59757995, 3.61539975, 4.13322401])`
	
	df = 14; loc = 4; $scale$ = 0.5
	t.stats(df, loc, $scale$, moments='mvsk')
	
	`Salida: (4.0, 0.2916666666666667, 0.0, 0.6)`
	
	`datos = [4.59757995, 3.61539975, 4.13322401]`
	t.fit(datos)
	
	`Salida: (37948107796.412155, 4.11540123604, 0.40117172698)`
	
	confianza = 0.95; df = 14; loc = 4; $scale$ = 0.5
	t.interval(confianza, df, loc, $scale$)
	
	`Salida: (2.9276066560415366, 5.072393343958463)`
		$$f(x,\upsilon) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{\pi\nu s^2}}\left(1+\frac{1}{\nu}\left(\frac{x-\bar{x}}{s}\right)^2\right)^{-\frac{\nu+1}{2}}$$
		
		x son los valores que va a tomar la variable aleatoria.
		$\bar{x} \rightarrow loc$ es la media muestral. Por defecto $loc = 0$.
		$s \rightarrow scale$ es equivalente a la desviación estándar. Por defecto $scale = 1$.
		$\upsilon \rightarrow df$ son los grados de libertad igual al tamaño de la muestra disminuido en 1 $(n-1)$.
		f es una probabilidad.
		size es el número de elementos.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

8	x = 600; *loc* = 0; *scale* = 2000
	expon.pdf(x, *loc*, *scale*)
	
	`Salida: 0.00037040911034085894`
	
	x = 600; *loc* = 0; *scale* = 2000
	expon.cdf(x, *loc*, *scale*)
	
	`Salida: 0.2591817793182821`
	
	f = 0. 2591817793182821; *loc* = 0; *scale* = 2000
	expon.ppf(f, *loc*, *scale*)
	
	`Salida: 600.0`
	
	loc = 0; *scale* = 2000; m = 3
	np.random.seed(10)
	expon.rvs(*loc*, *scale*, *size*=m)
	
	`Salida: array([2950.86889109, 41.94059421, 2008.32260078])`
	
	loc = 0; *scale* = 2000
	expon.stats(*loc*, *scale*, moments='mvsk')
	
	`Salida: (2000.0, 4000000.0, 2.0, 6.0)`
	
	datos = `[2950.86889109, 41.94059421, 2008.32260078]`
	expon.fit(datos, *floc*=0)
	
	`Salida: (0.0, 1667.0440286933335)`
	
	confianza = 0.95; *loc* = 0; *scale* = 2000
	expon.interval(confianza, *loc*, *scale*)
	
	`Salida: (50.63561596857979, 7377.758908227871)`
		$$f(x) = \frac{1}{\mu} e^{-\left(\frac{x-x_0}{\mu}\right)}$$
		
		x son los valores que va a tomar la variable aleatoria.
		$\mu \rightarrow scale$ es la media poblacional. Por defecto $scale = 1$.
		$x_0 \rightarrow loc$ es el valor inicial de x. Por defecto $loc = 0$.
		f es una probabilidad.
		size es el número de elementos.
		datos es una muestra que se distribuye exponencialmente.
		En *_.fit()*, si $x_0 \neq 0$, se debe borrar *floc*=0.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

9	x = 600; β = 1.5; loc = 0; $scale$ = 2000
	weibull_min.pdf(x, β, loc, $scale$)
	
	`Salida: 0.0003485459377373572`
	
	x = 600; β = 1.5; loc = 0; $scale$ = 2000
	weibull_min.cdf(x, β, loc, $scale$)
	
	`Salida: 0.1515267892198147`
	
	f = 0.1515267892198147; β = 1.5; loc = 0; $scale$ = 2000
	weibull_min.ppf(f, β, loc, $scale$)
	
	`Salida: 600.0000000000001`
	
	β = 1.5; loc = 0; $scale$ = 2000; m = 3
	np.random.seed(10)
	weibull_min.rvs(β, loc, $scale$, *size*=m)
	
	`Salida: array([2592.04940206, 152.08967015, 2005.54455953])`
	
	β = 1.5; loc = 0; $scale$ = 2000
	weibull_min.stats(β, loc, $scale$, moments='mvsk')
	
	`Salida: (1805.4906, 1502761.13926, 1.0720, 1.3904)`
	
	datos = [2592.04940206, 152.08967015, 2005.54455953]
	weibull_min.fit(datos, *floc*=0)
	
	`Salida: (1.157516063882655, 0, 1651.6711831119792)`
	
	confianza = 0.95; β = 1.5; loc = 0; $scale$ = 2000
	weibull_min.interval(confianza, β, loc, $scale$)
	
	`Salida: (172.44372549614465, 4774.848956100407)`
		$$f(x,\beta,\eta) = \frac{\beta}{\eta}\left(\frac{x-x_0}{\eta}\right)^{\beta-1} e^{-\left(\frac{x-x_0}{\eta}\right)^{\beta}}$$
		
		x son los valores que va a tomar la variable aleatoria.
		β es el parámetro de forma.
		$\eta \rightarrow scale$ es equivalente a la desviación estándar. Por defecto $scale = 1$.
		$x_0 \rightarrow loc$ es el valor inicial de x. Por defecto $loc = 0$.
		f es una probabilidad.
		size es el número de elementos.
		datos es una muestra que se distribuye de acuerdo con Weibull.
		En *_.fit()*, si $x_0 \neq 0$, se debe borrar *floc*=0.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

10	x = 600; α = 200; loc = 0; $scale$ = 3
	gamma.pdf(x, α, loc, $scale$)
	
	`Salida: 0.00939924256197369`
	
	x = 600; α = 200; loc = 0; $scale$ = 3
	gamma.cdf(x, α, loc, $scale$)
	
	`Salida: 0.5094034180072367`
	
	f = 0.5094034180072367; α = 200; loc = 0; $scale$ = 3
	gamma.ppf(f, α, loc, $scale$)
	
	`Salida: 600.0`
	
	α = 200; loc = 0; $scale$ = 3; m = 3
	np.random.seed(10)
	gamma.rvs(α, loc, $scale$, *size*=m)
	
	`Salida: array([657.23902087, 629.83591893, 625.7270207])`
	
	α = 200; loc = 0; $scale$ = 3
	gamma.stats(α, loc, $scale$, moments='mvsk')
	
	`Salida: (600.0, 1800.0, 0.1414213562373095, 0.03)`
	
	datos = `[657.23902087, 629.83591893, 625.7270207]`
	gamma.fit(datos, *floc*=0)
	
	`Salida: (2097.6220659954447, 0, 0.3039635517933118)`
	
	confianza = 0.95; α = 200; loc = 0; $scale$ = 3
	gamma.interval(confianza, α, loc, $scale$)
	
	`Salida: 519.7226480443719, 685.9582229490975)`
		$$f(x,\alpha,\beta)=\frac{(x-x_0)^{\alpha-1}}{\beta^{\alpha}\,\Gamma(\alpha)}e^{-\frac{x}{\beta}}$$
		
		x son los valores que va a tomar la variable aleatoria.
		α es el parámetro de forma.
		$x_0 \rightarrow loc$ es la media muestral. Por defecto $loc = 0$.
		$\beta \rightarrow scale$ es equivalente a la desviación estándar. Por defecto $scale = 1$.
		f es una probabilidad.
		size es el número de elementos.
		datos es una muestra que se distribuye de acuerdo con gamma.
		En *_.fit()*, si $x_0 \neq 0$, se debe borrar *floc*=0.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción

11	x = 1800; df = 2; loc = 0; $scale$ = 900
	chi2.pdf(x, df, loc, $scale$)
	
	`Salida: 0.00020437746731746792`
	
	x = 1800; df = 2; loc = 0; $scale$ = 900
	chi2.cdf(x, df, loc, $scale$)
	
	`Salida: 0.6321205588285577`
	
	f = `0.6321205588285577`; df = 2; loc = 0; $scale$ = 900
	chi2.ppf(f, df, loc, $scale$)
	
	`Salida: 1800.0`
	
	df = 2; loc = 0; $scale$ = 900; m = 3
	np.random.seed(10)
	chi2.rvs(df, loc, $scale$, *size*=m)
	
	`Salida: array([2655.78200198, 37.74653479, 1807.4903407])`
	
	df = 2; loc = 0; $scale$ = 900
	chi2.stats(df, loc, $scale$, moments='mvsk')
	
	`Salida: (1800.0, 3240000.0, 2.0, 6.0)`
	
	datos = [2655.78200198, 37.74653479, 1807.4903407]
	chi2.fit(datos, *floc*=0)
	
	`Salida: (1.258523377057727, 0, 1192.1427443080158)`
	
	confianza = 0.95; df = 2; loc = 0; $scale$ = 900
	chi2.interval(confianza, df, loc, $scale$)
	
	`Salida: (45.57205437172181, 6639.983017405084)`
		$$f(x,k) = \frac{1}{2^{k/2}\Gamma\left(\frac{k}{2}\right)}(x-x_0)^{\frac{k}{2}-1}e^{-\frac{x-x_0}{2}}$$
		x son los valores que va a tomar la variable aleatoria.
		$x_0 \rightarrow loc$ es el valor inicial de x. Por defecto $loc = 0$.
		$scale$ está relacionada con la dispersión de la distribución. Por defecto $scale = 1$.
		$k \rightarrow df$ son los grados de libertad.
		f es una probabilidad.
		size es el número de elementos.
		datos es una muestra que se distribuye de acuerdo con chi-cuadrado.
		En *_.fit()*, si $x_0 \neq 0$, se debe borrar *floc*=0.
		confianza es un porcentaje entre 0 y 1. El más utilizado es 0,95.

N°	Función	Descripción
12	datos = [4.66579325, 4.35763949, 3.22729985] *shapiro*(datos) Salida: ShapiroResult(statistic = 0. 0.9017996788024902, pvalue = 0.39125339167311535) Como el pvalue = 0.94 > 0.05, los datos se distribuyen normalmente	Realizar la prueba de Shapiro-Wilk para verificar la normalidad en la muestra "datos" para $n > 30$.
13	datos = [4.66579325, 4.35763949, 3.22729985] *kstest*(datos, '*norm*' , *args*= *(np.mean*(datos), *np.std*(datos))) Salida: KstestResult(statistic = 0.33784571608792285, pvalue = 0.7657051364545153, statistic_location = 4.35763949, statistic_sign=-1) Como el pvalue = 0.77 > 0.05, los datos se distribuyen normalmente	Realizar la prueba de Kolmogorov-Smirnov (KS) con la variante de Lilliefors para verificar la normalidad en la muestra "datos" para $n < 30$.
14	H0: $\mu_a = m$ La media de la muestra "a" es significativamente igual con la media de la población. H1: $\mu_a \neq m$ La media de la muestra "a" es significativamente diferente con la media de la población. a = [4.59757995, 3.61539975, 4.13322401] popmean = 4 *ttest_1samp*(a, popmean, *alternative*='two-sided') Salida: TtestResult(statistic = 0.40681364268729264, pvalue = 0.7235499444905745, df=2)	Realiza una prueba *t* para determinar si la media de una muestra (a) es igual a una media poblacional conocida (*popmean*). *alternative* indica la hipótesis alternativa a probar. Puede ser '*two-sided*' (por defecto), '*less*' (para una prueba de cola izquierda) o '*greater*' (para una prueba de cola derecha). *equal_var* indica si se asume igual varianza en ambas muestras. Puede ser '*True*' (por defecto) o '*False*' (se realiza la prueba *t* de *Welch*, que no asume igual

N°	Función	Descripción
	`Como el pvalue = 0.72 > 0.05, se rechaza H1 y se concluye que` **`no`** `existe suficiente evidencia para aseverar que La media de la muestra "a" es significativamente diferente con la media de población, con una confianza del 95%.`	varianza). *nan_policy* indica lo que ocurre si en una de las dos muestras existe por lo menos un valor *NaN*. Puede ser '*propagate*' (por defecto, la prueba no se realiza), '*raise*' (se indica que los valores *NaN* no están permitidos), o '*omit*' (se realiza la prueba omitiendo los valores *NaN*). *axis* indica el eje a lo largo del cual se realiza la prueba (por defecto *axis*=0).

15	H0: $\mu_1 = \mu_2$ El caso 1 con el caso 2 no tienen diferencias significativas que antes.
	H1: $\mu_1 \neq \mu_2$ El caso 1 con el caso 2 tienen diferencias significativas que antes.
	
	caso1 = [4.66579325, 4.35763949, 3.22729985]
	caso2 = [2.52061374, 3.03456492, 2.11538559]
	
	ttest_ind(caso1, caso2, *alternative*='*two-sided*')
	
	`Salida: TtestResult(statistic = 2.982973022997303, pvalue = 0.04061924564760709, df=4.0)`
	
	`Como el pvalue = 0.04 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que el caso 1 con el caso 2 tienen diferencias significativas, con una confianza del 95%.`
		Para la prueba *t* de dos muestras independientes.
		caso1 es la muestra de referencia.
		caso2 es la muestra sometida a la prueba.
		alternative indica la hipótesis alternativa a probar. Puede ser '*two-sided*' (por defecto), '*less*' (para una prueba de cola izquierda) o '*greater*' (para una prueba de cola derecha).
		equal_var indica si se asume igual varianza en ambas muestras. Puede ser '*True*' (por defecto) o '*False*' (se realiza la prueba t de Welch, que no asume igual varianza).
		nan_policy indica lo que ocurre si en una de las dos muestras existe por lo menos un valor *NaN*. Puede ser '*propagate*' (por defecto, la prueba no se realiza), '*raise*' (se indica que los valores *NaN* no están permitidos), o '*omit*' (se realiza la prueba omitiendo los valores *NaN*).
		axis indica el eje a lo largo del cual

N°	Función	Descripción

	H0: $\mu_1 < \mu_2$ El caso 1 es significativamente menor que el caso 2.
	H1: $\mu_1 > \mu_2$ El caso 1 es significativamente mayor que el caso 2.
	
	ttest_ind(caso1, caso2, *alternative*='*greater*')
	
	`Salida: TtestResult(statistic = 2.982973022997303, pvalue = 0.020309622823803546, df=4.0)`
	
	`Como el pvalue = 0.02 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que el caso 1 es significativamente mayor que el caso 2, con una confianza del 95%.`
		se realiza la prueba (por defecto *axis*=0).

16	H0: $\mu_d = \mu_a$ Después de la implementación no existe diferencias significativas que antes.
	H1: $\mu_d \neq \mu_a$ Después de la implementación existe diferencias significativas que antes.
	
	despues = [4.66579325, 4.35763949, 3.22729985]
	antes = [2.52061374, 3.03456492, 2.11538559]
	
	ttest_rel(despues, antes, *alternative*='*two-sided*')
	
	`Salida: TtestResult(statistic = 4.843983865126914, pvalue = 0.04007380570560517, df=2)`
	
	`Como el pvalue = 0.04 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que después de la`
		Para la prueba *t* de dos muestras pareadas o relacionadas.
		antes es la muestra de referencia.
		despues es la muestra sometida a la prueba.
		alternative indica la hipótesis alternativa a probar. Puede ser '*two-sided*' (por defecto), '*less*' (para una prueba de cola izquierda) o '*greater*' (para una prueba de cola derecha).
		nan_policy indica lo que ocurre si en una de las dos muestras existe por lo menos un valor *NaN*. Puede ser '*propagate*' (por defecto, la prueba no se realiza), '*raise*' (se indica que los valores *NaN* no están permitidos), o '*omit*' (se realiza la prueba omitiendo los valores *NaN*).
		axis indica el eje a lo largo del cual

N°	Función	Descripción

	`implementación existe diferencias significativas que antes, con una confianza del 95%.`
	
	H0: $\mu_d < \mu_a$ Después de la implementación existe una disminución significativa.
	H1: $\mu_d > \mu_a$ Después de la implementación existe un incremento significativo.
	
	ttest_rel(despues, antes, *alternative*='*greater*')
	
	`Salida: TtestResult(statistic = 4.843983865126914, pvalue = 0.020036902852802584, df=2)`
	
	`Como el pvalue = 0.02 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que después de la implementación existe un incremento significativo, con una confianza del 95%.`
		se realiza la prueba (por defecto *axis*=0).

17	H0: $\mu_1 = \mu_2$ El caso 1 con el caso 2 no tienen diferencias significativas que antes. H1: $\mu_1 \neq \mu_2$ El caso 1 con el caso 2 tienen diferencias significativas que antes. caso1 = [4.66579325, 4.35763949, 3.22729985] caso2 = [2.52061374, 3.03456492, 2.11538559] *ranksums*(caso1, caso2, *alternative*='two-sided') `Salida: RanksumsResult (statistic = 1.963961012123931, pvalue = 0.049534613435626706)`	Realizar la prueba de Wilcoxon-Mann-Whitney para dos muestras independientes. caso1 es la muestra de referencia. caso2 es la muestra sometida a la prueba. *alternative* indica la hipótesis alternativa a probar. Puede ser '*two-sided*' (por defecto), '*less*' (para una prueba de cola izquierda) o '*greater*' (para una prueba de cola derecha). *use_continuity* indica si se debe utilizar la corrección de continuidad en el cálculo de la estadística de prueba. Por defecto '*True*'.

N°	Función	Descripción
	Como el pvalue = 0.0495 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que el caso 1 con el caso 2 tienen diferencias significativas, con una confianza del 95%. H0: $\mu_1 < \mu_2$ El caso 1 es significativamente menor que el caso 2. H1: $\mu_1 > \mu_2$ El caso 1 es significativamente mayor que el caso 2. *ranksums*(caso1, caso2, *alternative*='*greater*') Salida: RanksumsResult (statistic = 1.963961012123931, pvalue = 0.024767306717813353) Como el pvalue = 0.0248 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que el caso 1 es significativamente mayor que el caso 2, con una confianza del 95%.	
18	H0: $\mu_d = \mu_a$ Después de la implementación no existe diferencias significativas que antes. H1: $\mu_d \neq \mu_a$ Después de la implementación existe diferencias significativas que antes. despues = [4.66579325, 4.35763949, 3.22729985] antes = [2.52061374, 3.03456492, 2.11538559] *wilcoxon* (despues, antes, *alternative*='two-sided')	Realizar la prueba de Wilcoxon-Mann-Whitney para dos muestras pareadas o relacionadas. antes es la muestra de referencia. despues es la muestra sometida a la prueba. Puede ser una constante. *alternative* indica la hipótesis alternativa a probar. Puede ser '*two-sided*' (por defecto), '*less*' (para una prueba de cola izquierda) o '*greater*' (para una prueba de cola derecha). *zero_method*= Indica cómo manejar los valores cero en las

N°	Función	Descripción
	Salida: WilcoxonResult(statistic = 0.0, pvalue=0.25) Como el pvalue = 0.25 > 0.05, se rechaza H1 y se concluye que **no** existe suficiente evidencia para aseverar que después de la implementación existe diferencias significativas que antes, con una confianza del 95%. H0: $\mu_d < \mu_a$ Después de la implementación existe una disminución significativa. H1: $\mu_d > \mu_a$ Después de la implementación existe un incremento significativo. *wilcoxon* (despues, antes, *alternative*='*greater*') Salida: WilcoxonResult (statistic = 1.963961012123931, pvalue = 0.024767306717813353) Como el pvalue = 0.0248 < 0.05, se rechaza H0 y se concluye que existe suficiente evidencia para aseverar que después de la implementación existe un incremento significativo, con una confianza del 95%.	diferencias entre las dos muestras. Puede ser '*wilcox*' (por defecto, clasifica los valores cero como empates) o '*zsplit*' (divide los empates por la mitad). *correction*= Indica si se debe aplicar una corrección de continuidad al cálculo de la estadística de prueba. Por defecto '*False*'. *mode*= Controla cómo se realiza el cálculo. Pueden ser '*auto*' (Por defecto, que selecciona automáticamente el método más apropiado según el tamaño de las muestras), '*exact*', '*approx*' o '*asymp*'.
19	*kstest*(datos, '*distribucion*', *args*=parametros) datos = [2592.04940206, 152.08967015, 2005.54455953] *kstest*(datos, '*weibull_min*', *args* = *weibull_min.fit*(datos, *floc*=0))	Realizar la prueba de Kolmogorov-Smirnov (KS), que es una prueba no paramétrica para determinar si una muestra de datos sigue una distribución de probabilidad específica. datos es la muestra que se desea avaluar.

N°	Función	Descripción

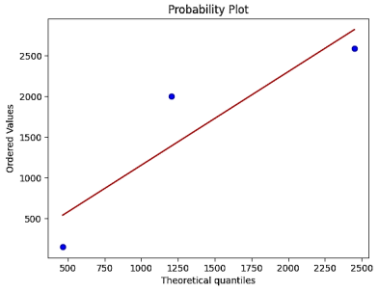
	`Salida: KstestResult(statistic = 0.380721425818675, pvalue = 0.648195262741535, statistic_location = 2005.54455953, statistic_sign = -1)`
	
	`Como el pvalue = 0.645 > 0.05, los datos se distribuyen de acuerdo con Weibull.`
		distribucion es la distribución que se desea evaluar, como: *'norm'*, *'expon'*, *'weibull_min'*, *'gamma'*, entre otras.
		parametros son los parámetros de la distribución que se desea evaluar. Pueden encontrarse con la función *fit*():
		parametros = *stats*.distribucion.*fit*(datos)

20	x = [1, 2, 3, 4, 5]
	y = [2, 4, 6, 8, 9]
	
	pearsonr(x, y)
	
	`Salida: PearsonRResult(statistic = 0.993883734673619, pvalue = 0.0005736731093321745)`
	
	`Como el pvalue = 0.0006 < 0.05, los datos se distribuyen linealmente.`
		Calcula el coeficiente de correlación "r" de Pearson y el p valor para probar si $pendiente \neq 0$.

21	x = [1, 2, 3, 4, 5] y = [2, 4, 6, 8, 10] *linregress*(x, y) `Salida: LinregressResult (slope = 2.0, intercept = 0.0, rvalue = 1.0, pvalue = 1.200421754876140e-30, stderr = 0.0, intercept_stderr = 0.0)`	Realiza una regresión lineal simple y calcula: La pendiente de la línea de regresión (*slope*). El intercepto de la línea de regresión (*intercept*). El coeficiente de correlación "r" de Pearson entre x e y (*rvalue*). El p valor para probar si $pendiente \neq 0$ (*pvalue*). El error estándar de la pendiente estimada *stderr*.
22	*from scipy.optimize import curve_fit*	Devuelve una tupla con 2 elementos *popt* (arreglo de parámetros estimados del

N°	Función	Descripción

	# Modelo potencial:
	def *func*(x, b0, b1):
	  return b0 * *np.power*(x, b1)
	
	x = [1, 2, 3, 4, 5]
	y = [0.5, 2, 4.5, 8, 12.5]
	
	curve_fit(*func*, x, y)
	
	`Salida:`
	`array([0.5, 2. ])`
	
	# Modelo exponencial:
	def func(x, b0, b1):
	  *return b0 * np.exp(b1 * x)*
	
	x = [1, 2, 3, 4, 5]
	y = [1, 2, 4, 8, 16]
	
	curve_fit(*func*, x, y)
	
	`Salida:`
	`array([0.5, 0.69314718])`
		modelo) y *pcov* (matriz de covarianza de los parámetros estimados).
		func es el modelo al que se deseas ajustar los datos (potencial, exponencial, entre otros). "*Def func*(x, parámeros)".
		x es un arreglo *np.array* con los valores de la variable independiente.
		y es un arreglo *np.array* con los valores de la variable dependiente.
		p0 es una lista o arreglo de valores iniciales para los parámetros del modelo. Por defecto 1.0.
		sigma es una lista o arreglo con las incertidumbres de los valores en "y".
		absolute_sigma indica si las incertidumbres en sigma son absolutas (*True*) o relativas (*False*). Por defecto es *False*.
		bounds es una tupla que define los límites inferior y superior para los parámetros. Por defecto los parámetros no tienen límites.
		method es el método de optimización que se utiliza para ajustar la función de modelo a los datos. Puede ser '*lm*' (*Least Squares*), '*trf*' (*Trust Region Reflective*) o una instancia de una clase *OptimizeResult*. Por defecto '*lm*'.
		jac es una función que calcula la matriz jacobiana de *func*. Por defecto no se proporciona y se calcula numéricamente.

N°	Función	Descripción
23	*import matplotlib.pyplot as plt* datos = [2592.04940206, 152.08967015, 2005.54455953] *probplot*(datos, *sparams*= weibull_min.fit(datos, floc=0), *dist*='*weibull_min*', *plot*=*plt*) `Salida:`  	Crea el diagrama Q-Q de una muestra. datos es la muestra que se desea representar en el diagrama Q-Q. *sparams* son los parámetros de forma específicos de la distribución, si *dist*='*norm*' no es necesario proporcionarlos. *dist* es la distribución que se utiliza para calcular los cuantiles teóricos. Pueden ser '*expon*', '*uniform*', '*weibull_min*', entre otros. Por defecto '*norm*'. fit indica si se debe ajustar la distribución. Por defecto *True*, los parámetros se encuentran por máxima verosimilitud). *plot* se emplea únicamente para graficar dentro de un objeto de ejes de Matplotlib. *rvalue* indica el coeficiente de determinación "R^2" en el gráfico Q-Q. Por defecto *False*.

Nota. Los autores**.**

1.5.5 Librería *Sklearn*

Scikit-Learn o *Sklearn* en su forma abreviada es una biblioteca de aprendizaje automático (*machine learning* en inglés), ciencia de datos y el análisis predictivo, de código abierto en *Python* que ofrece una amplia gama de herramientas para

tareas de clasificación, regresión, agrupación, selección de características, evaluación de modelos, preparación de datos y la evaluación de modelos (Scikit-Learn, 2023), de las cuales en la presente obra se utiliza la sublibrería *sklearn.impute* para el tratamiento de datos faltantes. La Tabla 1.6 indica algunas funciones para este fin.

Tabla 1.6

Principales funciones de Sklearn.impute.

N°	Función	Descripción
1	*from sklearn.impute import SimpleImputer* *imputer* = *SimpleImputer*(*strategy*='mean') datos2 = *imputer.fit_transform*(datos)	Proporciona estrategias básicas para la imputación de valores faltantes utilizando la media de los datos (*'mean'*). *strategy* puede ser igual a otras alternativas como: (*strategy*='*median*') utiliza la mediana. (*strategy*='*most_frequent*') utiliza el valor más frecuente. (*strategy*='*constant*', *fill_value*=n) utiliza "n" que debe ser una constante.
2	*from sklearn.impute import KNNImputer* *imputer* = *KNNImputer* (*n_neighbors*=n) datos2 = *imputer.fit_transform*(datos)	Utiliza el algoritmo K-Nearest Neighbors (K-NN) para imputar valores faltantes utilizando el promedio de los "n" valores vecinos más cercanos.

N°	Función	Descripción
3	*from sklearn.experimental import enable_iterative_imputer* *from sklearn.impute import IterativeImputer* *imputer* = *IterativeImputer*(*max_iter*=10, *random_state*=0) datos2 = *imputer*.*fit_transform*(datos)	Imputa los valores faltantes mediante un enfoque iterativo que considera todas las características y utiliza modelos de regresión.

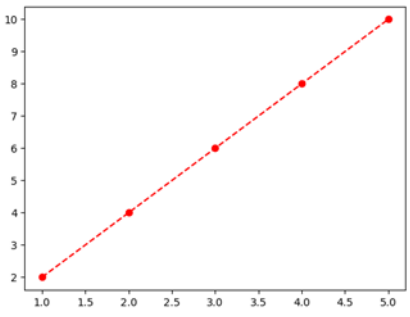
Nota. Los autores**.**

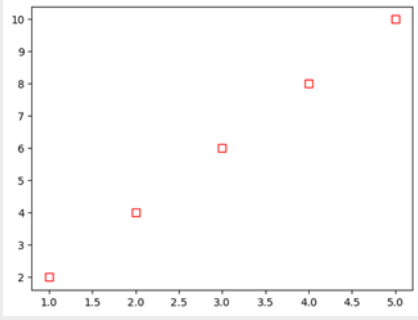
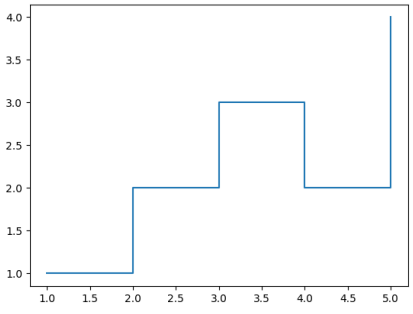
1.5.6 Librería *Matplotlib*

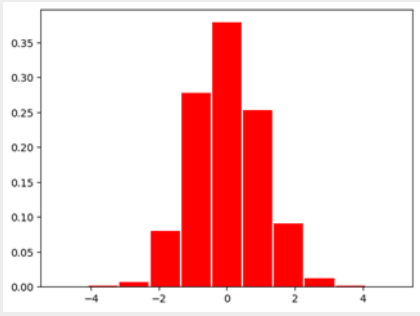
Es ampliamente utilizada para la creación de gráficos, visualizaciones de datos y presentación de resultados científicos. Ofrece una variedad de funciones para generar gráficos de líneas, barras, dispersión, histogramas, entre otros, con una gran flexibilidad para la personalización (Matplotlib, 2023). La Tabla 1.7 indica varias funciones habituales de esta librería.

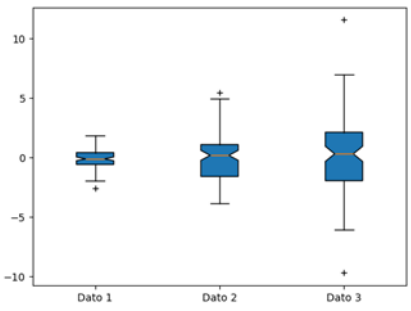
Tabla 1.7

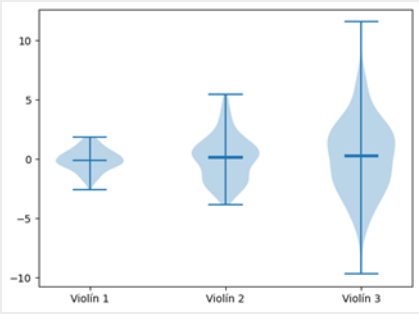
Principales funciones de Matplotlib.

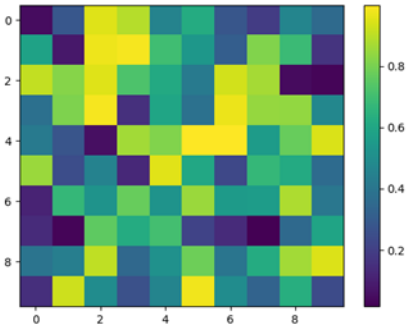
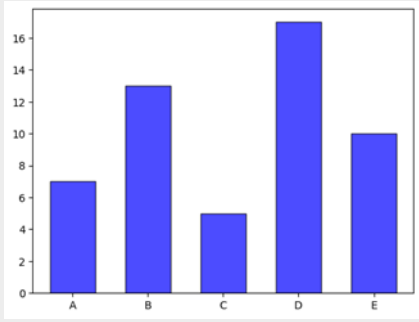
N°	Función	Descripción
1	*import matplotlib.pyplot as plt*	Importa la librería *Matplotlib*
2	*plt.plot*(x, y, 'r-.o') x = [1, 2, 3, 4, 5] y = [2, 4, 6, 8, 10] *plt.plot*(x, y, *color*='*red*', *linestyle*='--', *marker*='o', *label*='Nombre') 	Traza líneas o puntos en un gráfico. x es una lista con los valores de la variable independiente. y es una lista con los valores de la variable dependiente. *color* es el color de la línea y marcador ('red', 'blue', 'black', entre otros). *linestyle* estilo de la línea ('-', '--', '-.', ':', entre otros). *marker* tipo de marcador ('o', '^', 's', entre otros). 'r-.o' reemplaza a *color*, *linestyle* y *marker*. El primer signo corresponde al color (en este ejemplo "r" rojo), el segundo signo corresponde al estilo de la línea (en este ejemplo "-." líneas y puntos), y el último dígito corresponde al marcador (en este ejemplo "o" círculos). *linewidth* ancho de la línea. *markerfacecolor* color del marcador. *mfc* color del centro de la marca. *mec* color del contorno de la marca. *label* nombre que aparece en la etiqueta de la leyenda.
3	x = [1, 2, 3, 4, 5] y = [2, 4, 6, 8, 10]	Elabora gráficos de dispersión. x es una lista con los valores de la variable independiente.

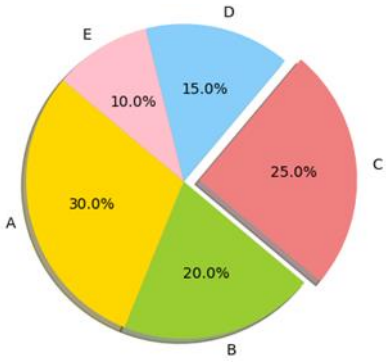
N°	Función	Descripción
	plt.scatter(x, y, *color*='*white*', *edgecolors*='*red*', *marker*='s', s=n, *label*='Nombre') 	y es una lista con los valores de la variable dependiente. *color* color de la línea y marcador ('red', 'blue', 'black', entre otros). *edgecolors* color del centro de los marcadores. *marker* tipo de marcador ('o', '^', 's', '+', '*', entre otros). s es el tamaño "n" del marcador. En este ejemplo $s = 60$. *label* nombre que aparece en la etiqueta de la leyenda.
4	x = [1, 2, 3, 4, 5] y = [1, 2, 3, 2, 4] *plt.step*(x, y, *where* = '*post*') 	Crea gráficos de escalones x es una lista con los valores de la variable independiente. y es una lista con los valores de la variable dependiente. *where* ubica los escalones. Pueden ser '*pre*' (se ubican antes de "x"), '*post*' (se ubican después de "x") o '*mid*' (se ubican en el punto medio entre los valores de "x"). *color* es el color de la línea y marcador ('red', 'blue', 'black', entre otros). *linestyle* estilo de la línea ('-', '--', '-.', ':', entre otros). *marker* tipo de marcador ('o', '^', 's', '+', '*', entre otros) *linewidth* ancho de la línea. *markerfacecolor* color del marcador. *mfc* color del centro de la marca. *mec* color del contorno de la marca. *label* nombre que aparece en la etiqueta de la leyenda.

N°	Función	Descripción
5	*import numpy as np* *np.random.seed*(42) x = *np.random.normal*(0, 1, 1000) q = *round*(1 + np.log2(1000)) x1 = -5 x2 = 5 a = 0 b = 1 *plt.hist*(x, *bins*=q, *range*=(x1, x2), *density*=*True*, *weights*=*None*, *cumulative*=*False*, *bottom*=a, *histtype*='*bar*', *align*='*mid*', *orientation*='*vertical*', *rwidth*=b, *log*=*False*, *color*='*red*', *label*='nombre', *stacked*=*False*, *edgecolor*='*white*') 	Crea histogramas. x lista con los valores que se utilizan para crear el histograma. *bins* es el número "q" de barras del histograma. Por defecto $bins = 10$. Se recomienda que se calcule con la ley de Sturges ($q = 1 + log_2(n)$), donde n es el tamaño de la muestra. *range* es el rango de valores del eje x (x1, x2) que se incluyen en el histograma. Por defecto se utiliza el rango completo de "x". *density* si es *True*, la altura de las barras se estandarizan (histograma de frecuencias relativas). *weights* es un vector pesos de *len*(x) elementos por el que se multiplica cada elemento de "x". *cumulative* si es *True*, se crea un histograma de frecuencias acumulativas. *bottom* es la separación "a" de las barras con el eje de las abscisas. *histtype* es el tipo de histograma. Puede ser '*bar*' (predeterminado), '*barstacked*', '*step*', 'stepfilled', o '*step*'. *align* determina la alineación de las barras con respecto a los valores de "x". Puede ser '*left*', '*mid*' (predeterminado), o '*right*'. *orientation* es la orientación del histograma. Puede ser '*vertical*' (predeterminado) o '*horizontal*'. *rwidth* es el ancho porcentual "b" de las barras. Puede ser un

N°	Función	Descripción
		número entre 0 y 1. *log* si es *True*, el eje de las ordenadas se pasa a escala logarítmica. Por defecto *False*. *color* determina el color de las barras del histograma. *label* etiqueta 'nombre' que se utiliza para identificar el histograma en una leyenda. *stacked* si es *True*, los histogramas se apilan en lugar de superponerse. Por defecto *False*. *edgecolor* es el color de las líneas de las barras.
6	*import numpy as np* *np.random.seed*(42) datos = [*np.random.normal*(0, *std*, 100) *for std in range*(1, 4)] nombres = ['Dato 1', 'Dato 2', 'Dato 3'] *plt.boxplot*(datos, labels=nombres, *vert*=*True*, *patch_artist*=*True*, *notch*=*True*, *sym*='k+') 	Elabora diagramas de caja. datos es un arreglo o un *DataFrame* que va a ser representado en el diagrama de cajas. *labels* es el vector con los nombres de cada caja del diagrama. *vert* si es *True* (predeterminado), el diagrama de cajas se orienta verticalmente. *patch_artist* si es *False* (predeterminado), las cajas no se rellenan con colores. *notch* si es *False* (predeterminado), no se realiza un corte en la mediana. *sym* define el color y símbolo que se utiliza para representar los valores atípicos (en este ejemplo 'k+' genera signos "+" negros).

N°	Función	Descripción
7	*import numpy as np* *np.random.seed*(42) datos = [*np.random.normal*(0, std, 100) for std in range(1, 4)] posiciones = [1, 2, 3] nombres = ['Violín 1', 'Violín 2', 'Violín 3'] *plt.violinplot*(datos, *showmeans=True*, *showmedians=True*, *showextrema=True*) *plt.xticks*(posiciones, nombres) 	Elabora diagramas de violín. datos es un arreglo o un *DataFrame* que va a ser representado en el diagrama de cajas. *showmeans* si es *True* muestra una línea para la media de cada conjunto de datos. *showmedians* si es *True* muestra una línea para la mediana de cada conjunto de datos. *showextrema* si es *True* muestra líneas para los valores mínimos y máximos de cada conjunto de datos. plt.xticks(posiciones, nombres) agrega nombres a cada uno de los violines.
8	*import numpy as np* datos = *np.random.rand*(10, 10) *plt.imshow*(datos, *cmap='viridis'*, *interpolation='nearest'*, *origin='upper'*, *aspect='auto'*) *plt.colorbar*()	**Elabora mapas de calor** datos es un arreglo o un *DataFrame* que va a ser representado en el diagrama de cajas. *cmap* indica el mapa de colores que se utiliza para representar los valores de la matriz (en este ejemplo "*viridis*"). *Interpolation* es el método de interpolación para suavizar la representación de los datos (en este caso '*nearest*').

N°	Función	Descripción
		*origin* indica la posición del origen en la esquina superior del gráfico (en este ejemplo '*upper*'). *aspect* determina el aspecto del gráfico, que puede ser '*auto*', '*equal*', o un número que controle la relación entre los ejes X e Y. *plt.colorbar*() coloca la barra
9	categorias = ['A', 'B', 'C', 'D', 'E'] valores = [7, 13, 5, 17, 10] *plt.bar*(categorias, valores, *color*='*blue*', *alpha*=0.7, *edgecolor*='*black*', *width*=0.6, *label*='Valores') 	Crea gráficos de barras. categorias son las etiquetas de las categorías (eje X). valores son los valores de las barras (eje Y). *color* determina el color de las barras (en este ejemplo '*blue*'). *alpha* es un valor entre 0 y 1 que determina la transparencia de las barras (en este ejemplo 0.7). *edgecolor* determina el color de los bordes de las barras (en este ejemplo '*black*'). *width* es un valor entre 0 y 1 que determina el ancho de las barras (en este ejemplo 0.6). *label* es el nombre de la etiqueta (en este ejemplo 'Valores').
10	sizes = [30, 20, 25, 15, 10] etiquetas = ['A', 'B', 'C', 'D', 'E'] colores = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'pink'] separaciones = (0, 0, 0.1, 0, 0) *plt.pie*(*sizes*, *labels*=etiquetas, *colors*=colores, *autopct*='*%1.1f%%*', *startangle*=140, *shadow*=*True*, *explode*=separaciones)	Crea gráficos de pastel. *sizes* es una lista que contiene los tamaños (valores) de cada porción. *labels* son las etiquetas para cada porción del pastel (en este ejemplo es la lista *etiquetas*). *colors* especifica los colores para cada porción (en este ejemplo, es la lista *colores*).

N°	Función	Descripción
		*autopct* agrega porcentajes en cada porción (en este ejemplo '%1.1f%%' porcentajes con 1 decimal). *startangle* determina el ángulo desde donde se inicia el gráfico (en este ejemplo 140 grados). *shadow* si es *True*, agrega sombras al gráfico. *explode* determina cuánto separar cada porción del pastel (por ejemplo, es la tupla $separaciones = (0.1, 0, 0, \cdots, 0)$ donde la 1ra porción se separa un 10%).
11	*plt.xlabel*('Nombre') *plt.ylabel*('Nombre') *plt.title*('Nombre')	Etiqueta los ejes "x" y "y" respectivamente y agrega un título al gráfico.
12	*plt.legend*()	Agrega leyendas a los gráficos.
13	*plt.grid*()	Activa las líneas de la cuadrícula en los 2 ejes del gráfico.
14	*plt.show*()	Muestra el gráfico en la ventana actual.
15	*plt.annotate*('Texto', *xy*=(x_punto, y_punto), *xytext*=(x_text, y_text), *arrowprops=dict*(*arrowstyle*='->', *connectionstyle*='arc3,rad=0.5'))	Agregar anotaciones o etiquetas en puntos específicos en un gráfico. *'Texto'* es el texto que se desea colocar en el gráfico. *xy* son las coordenadas (x_punto, y_punto) de la punta de la flecha. *xytext* son las coordenadas (x_text, y_text) del texto de la anotación. *arrowprops=dict(arrowstyle='->', connectionstyle='arc3, rad=0.5'))*

N°	Función	Descripción
		son las propiedades de la flecha que apunta desde el texto de la anotación hasta el punto.
16	*plt.text*(*x_text*, *y_text*, 'Texto', *fontsize*=12, *color*='*red*', *ha*='*center*', *va*='*bottom*')	Agrega texto en una ubicación específica dentro de un gráfico. *x_text* y *y_text* son las coordenadas del texto. 'Texto' es el texto que se desea mostrar en las coordenadas especificadas. *fontsize* es el tamaño de fuente del texto (en este ejemplo 12). *color* es el color del texto (en este ejemplo '*red*'). *ha*='*center*' y *va*='*bottom*' es la alineación horizontal y vertical del texto (en este ejemplo, el texto se alinea horizontalmente en el centro y verticalmente en la parte inferior.
17	*plt.xticks*(marcas_x, etiquetas_x) *plt.xticks*(*range*(1, *len*(*labels*) + 1), *labels*)	Personaliza las marcas y etiquetas en el eje X de un gráfico como los de bloque o de violín. marcas_x es una lista con los valores de las marcas del eje X. etiquetas_x es una lista con los str (cadenas de texto) de las etiquetas del eje X.
18	*plt.yticks*(marcas_y, etiquetas_y)	Personaliza las marcas y etiquetas en el eje Y de un gráfico. marcas_y es una lista con los valores de las marcas del eje Y. etiquetas_y es una lista con los str (cadenas de texto) de las etiquetas del eje Y.

N°	Función	Descripción
19	*plt.savefig*('nombre.png')	Guarda la figura en el mismo directorio donde está el archivo de *Python*, en diferentes formatos como PNG, JPEG, PDF, entre otros (en este ejemplo 'nombre.png').
20	*fig*=*plt.figure*()	Crea una nueva figura. *fig* es el nombre del objeto *plt.figure*().
21	*fig.add_subplot*(*nrows*, *ncols*, *index*)	Agrega ejes a una figura. *fig* es el nombre del objeto *plt.figure*(). *nrows* es el número de filas. *ncols* es el número de columnas. *index* es la posición del subgráfico. Inicia en 1, en orden de izquierda a derecha y de arriba hacia abajo.
22	*fig*, *axes* = *plt.subplots*(*nrows*=n, *ncols*=m, *figsize*=(a, b))	Crea múltiples subgráficos en una sola figura *fig* es el nombre del objeto *plt.subplots*(). *axes* es un arreglo que representa a un subgráfico. *nrows* es el "n" número de filas. *ncols* es el "m" número de columnas. *figsize* determina el tamaña de la figura.
23	*plt.tight_layout()*	Ajusta el espaciado entre los subgráficos.
24	*axes*[*i*, *j*].*set_xlabel*('Nombre') *axes*[*i*, *j*].*set_ylabel*('Nombre') *axes*[*i*, *j*].*set_title*('Nombre')	Etiqueta los ejes "x" y "y" respectivamente y agrega un título al gráfico. *axes* es un arreglo que representa a un subgráfico. *i* es el índice de la fila que ubica al subgráfico. Inicia en 0. *j* es el índice de la columna que ubica al subgráfico. Inicia en 0.

N°	Función	Descripción
25	*mpl_toolkits.mplot3d*	Crea gráficos de superficie, gráficos de dispersión 3D y más.

Nota. Los autores**.**

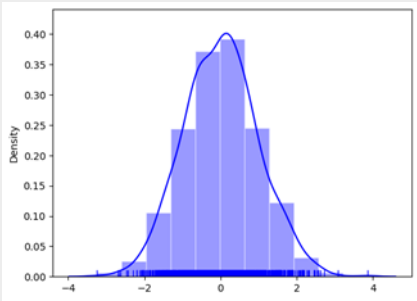
1.5.7 Librería *Seaborn*

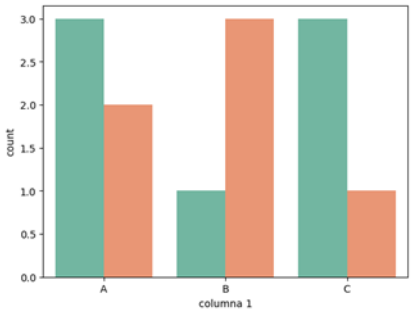
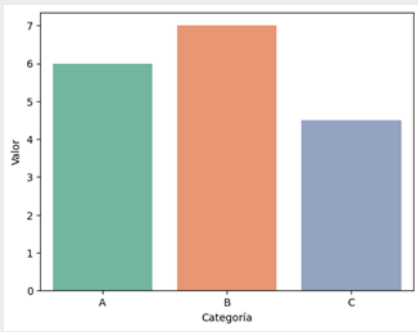
Es una librería de visualización de datos que se basa en *Matplotlib* y que integra de forma natural a *DataFrame* de *Pandas*, proporcionando una interfaz de alto nivel para crear gráficos estadísticos atractivos como histogramas, diagramas de dispersión, diagramas de caja, gráficos de violín y mapas de calor, entre otros (Seaborn, 2022). La Tabla 1.8 indica sus principales funciones.

Tabla 1.8

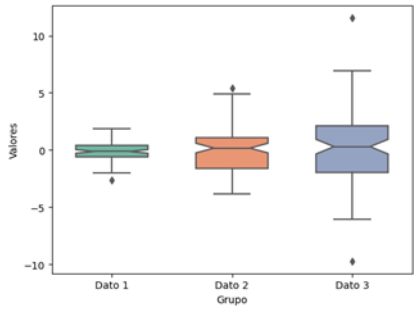
Principales funciones de Seaborn.

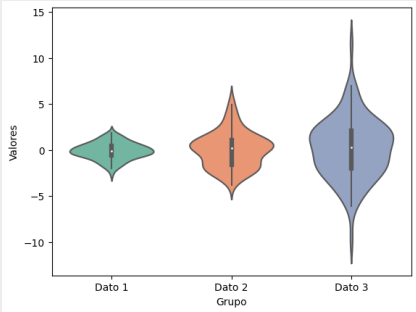
N°	Función	Descripción
1	*import seaborn as sns*	Importa la librería *Seaborn* datos debe ser un objeto *DataFrame*.
2	*sns.set_style*('*darkgrid*')	Permite establecer el estilo de los gráficos. Puede ser "*darkgrid*", "*whitegrid*", "*dark*", "*white*" y "*ticks*".
3	*import numpy as np* *np.random.seed*(42)	Permite crear histogramas. datos que deseas visualizar en el histograma.

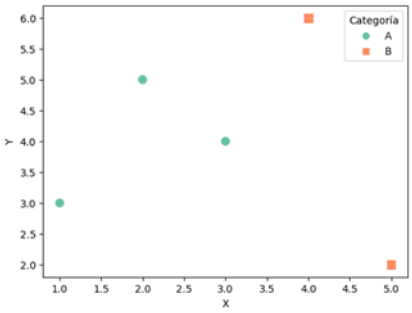
N°	Función	Descripción
	datos = *np.random.normal*(0, 1,1000) q = *round*(1 + np.log2(1000)) *sns.distplot*(datos, *bins*=q, hist=True, *kde*=True, *rug*=True, *color*='blue', *hist_kws* = {'*edgecolor*': '*white*'}, *label*='Distribución Normal')  	*bins* es la cantidad de columnas del histograma. Se recomienda que se calcule con la ley de Sturges ($q = 1 + log_2(n)$), donde n es el tamaño de la muestra. *hist* si es *True*, se muestra el histograma. *kde* si es true, se muestra la estimación de la función de densidad. *rug* si es True, se muestra los marcadores en el eje X para cada punto de datos. *color* determina el color del gráfico. *hist_kws* = {'edgecolor': 'white'} se emplea para que las líneas de las barras se hagan blancas. *label* es la etiqueta para la leyenda del gráfico. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura.
4	*import pandas as pd* datos = *pd.DataFrame*({ 'columna 1': ['A', 'B', 'C', 'A', 'A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C'], 'columna 2': [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1] }) *sns.countplot*(*x*='columna 1', *data*=datos, *hue*='columna 2', *palette*='*Set2*', *orient* ='*v*')	Crea gráficos de barras de la frecuencia de una variable categórica. x o y son los datos que se desea visualizar. Si se proporciona "x" el gráfico se dibuja verticalmente y si se proporciona "y" el gráfico se dibuja horizontalmente. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *order*, *hue_order* es un vector con los nombres de las columnas del *DataFrame* que se va a

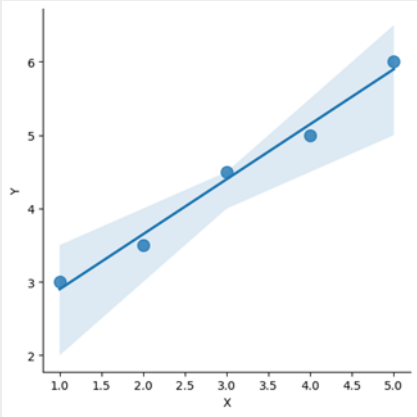
N°	Función	Descripción
		representar en el orden en el que se desea las columnas. *palette* es la paleta de colores que se utiliza para colorear las barras. *orient* orienta el gráfico. Puede ser "v" (vertical) o "h" (horizontal). *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como, *saturation*, *color*, *linewidth* y *edgecolor*.
5	*import pandas as pd* datos = *pd.DataFrame*({ 'Categoría': ['A', 'B', 'C', 'A', 'B', 'C'], 'Valor': [4, 9, 6, 8, 5, 3] }) *sns.barplot*(x='Categoría', y='Valor', *data*=datos, *palette*='*Set2*', *ci*=*None*) 	Crea gráficos de barras de los valores de una variable numérica relacionada con una variable categórica. x o y son los datos que se desea visualizar. Si los valores van en "y" el gráfico se dibuja verticalmente y si van en "x" el gráfico se dibuja horizontalmente. En el otro eje va la variable categórica. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *order*, *hue_order* es un vector con los nombres de las columnas del *DataFrame* que se va a representar en el orden en el que se desea las columnas. *estimator* es la función que se utiliza para calcular el valor que se muestra en cada barra. Puede ser *np.mean* (predeterminado), *np.median*, *np.sum*, u otra función personalizada.

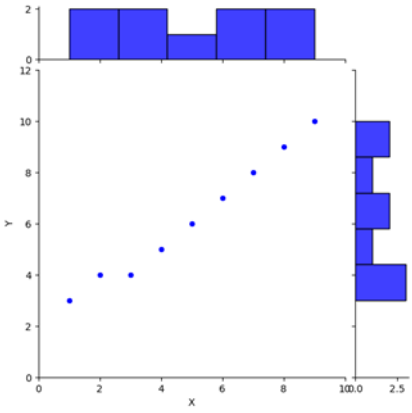
N°	Función	Descripción
		palette es la paleta de colores que se utiliza para colorear las barras. *orient* orienta el gráfico. Puede ser "v" (vertical) o "h" (horizontal). *ci* Nivel de confianza para la línea de regresión. Puede ser *None* (predeterminado), *'sd'*, *'boot'*, o un valor numérico. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como, *saturation*, *color*, *linewidth* y *edgecolor*.
6	*import numpy as np* *import pandas as pd* *np.random.seed*(42) datos = [*np.random.normal*(0, *std*, 100) *for std in range*(1, 4)] datos = *pd.DataFrame*({ 'Valores': *np.hstack*((datos[0], datos[1], datos[2])), 'Grupo': ['Dato 1']*100 + ['Dato 2']*100 + ['Dato 3']*100 }) *sns.boxplot*(x='Grupo', y='Valores', *data*=datos, *notch* =*False*, *palette*='*Set2*', *width*=0.6)	Genera diagramas de caja y bigotes. x o y son los datos que se desea visualizar. Si los valores van en "y" el gráfico se dibuja verticalmente y si van en "x" el gráfico se dibuja horizontalmente. En el otro eje va la variable categórica. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *order*, *hue_order* es un vector con los nombres de las columnas del *DataFrame* que se va a representar en el orden en el que se desea las columnas. *palette* es la paleta de colores que se utiliza para colorear las barras. *width* determina el ancho de las cajas en el gráfico ($0 < n < 1$). *orient* orienta el gráfico. Puede ser "v" (vertical) o "h" (horizontal).

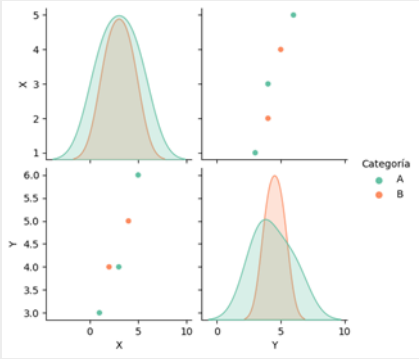
N°	Función	Descripción
		*notch* si es *False* (predeterminado), no se realiza un corte en la mediana. *sym* define el color y símbolo que se utiliza para representar los valores atípicos (en este ejemplo 'k+' genera signos "+" negros). *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como, *saturation*, *fliersize*, *color*, *linewidth* y *edgecolor*.
7	*import numpy as np* *import pandas as pd* *np.random.seed*(42) datos = [*np.random.normal*(0, *std*, 100) *for std in range*(1, 4)] datos = *pd.DataFrame*({ 'Valores': *np.hstack*((datos[0], datos[1], datos[2])), 'Grupo': ['Dato 1']*100 + ['Dato 2']*100 + ['Dato 3']*100 }) *sns.violinplot*(x='Grupo', y='Valores', *data*=datos, *palette*='Set2', *inner*='box', *split*=True, *scale*='width', *width*=0.7)	Crea gráficos de violín. x o y son los datos que se desea visualizar. Si los valores van en "y" el gráfico se dibuja verticalmente y si van en "x" el gráfico se dibuja horizontalmente. En el otro eje va la variable categórica. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *split* si es True, divide los violines cuando se utiliza el parámetro *hue*. *order*, *hue_order* es un vector con los nombres de las columnas del *DataFrame* que se va a representar en el orden en el que se desea las columnas. *palette* es la paleta de colores que se utiliza para colorear las barras. *inner* determina cómo se muestran los elementos internos del violín: *None* (predeterminado,

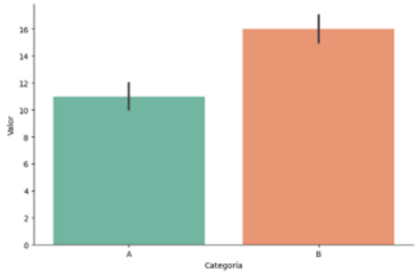
N°	Función	Descripción
		no muestra elementos internos), '*box*' (muestra una caja dentro del violín), '*quartile*' (similar a '*box*', pero sin la línea de la mediana), '*point*' (muestra puntos individuales para cada observación) y '*stick*' (muestra palitos verticales para cada observación dentro del violín). *scale* determina cómo se escalan los violines en el eje de ancho. '*area*' (predeterminado, los violines más delgados representan distribuciones con mayor densidad de datos), '*count*' (los violines más delgados representan categorías con menos observaciones) y '*width*' (ancho fijo para todos los violines, independientemente de la densidad de datos. Debes proporcionar un valor numérico para este caso *width*=n, $0 < n < 1$). *sym* define el color y símbolo que se utiliza para representar los valores atípicos (en este ejemplo 'k+' genera signos "+" negros). *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como, *saturation*, *fliersize*, *color*, *linewidth* y *edgecolor*.
8	*import pandas as pd* datos = *pd.DataFrame*({ 'X': [1, 2, 3, 4, 5],	Realiza gráficos de dispersión. x o y son los datos de las variables que se desean visualizar. *data* es el *DataFrame* que contiene los datos.

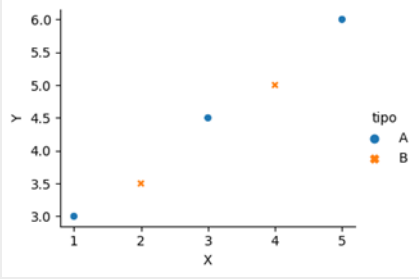
N°	Función	Descripción
	'Y': [3, 5, 4, 6, 2], 'Categoría': ['A', 'A', 'A', 'B', 'B'] }) *sns.scatterplot*(x='X', y='Y', *data*=datos, *hue*='Categoría', *palette*='*Set2*', *style*='Categoría', *markers*=['o', 's'], *s*=90)  	*hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *style* es una variable categórica opcional que se utiliza para cambiar el estilo de las marcas (Puede ser la misma utilizada en el parámetro *hue*). *size* es una variable numérica opcional que se utiliza para cambiar el tamaño de las marcas. *palette* es la paleta de colores que se utiliza para colorear las barras. *hue_order*, *style_order*, *size_order* es un vector con los nombres de las columnas del *DataFrame* que se va a representar en el orden en el que se desea las columnas. *markers* es una lista con los tipos de marcadores personalizados. *sizes* es el rango de tamaños de los marcadores. *alpha* es un valor entre 0 y 1 que determina la transparencia de los marcadores. *edgecolor* es el color del borde de los marcadores. *linewidth* es el ancho del borde de los marcadores. *legend* determina la visibilidad de la leyenda (por defecto, *True*). Existen otros argumentos para personalizar la apariencia del gráfico, como *xlabel*, *ylabel*, *title*, *xlim* y *ylim*.
9	*import pandas as pd*	Crea un gráfico de regresión lineal junto con su gráfico de dispersión.

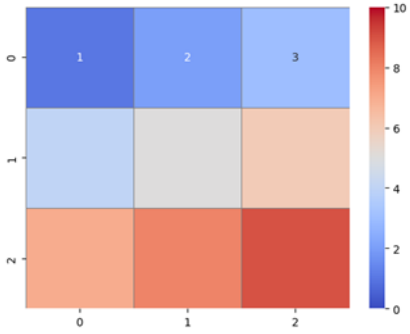
N°	Función	Descripción
	datos = *pd.DataFrame*({ 'X': [1, 2, 3, 4, 5], 'Y': [3, 3.5, 4.5, 5, 6] }) *sns.lmplot*(x='X', y='Y', *data*=datos, *hue*=*None*, *palette*='*Set2*', *scatter_kws*={"s": 100}, *ci*=*None*)  	x o y son los datos de las variables que se desean visualizar. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *col* y *row* son variables categóricas opcionales que dividen los datos en múltiples subgráficos (columnas y filas) según estas variables. *palette* es la paleta de colores que se utiliza para colorear las barras. *scatter_kws* y *line_kws* son diccionarios de argumentos que se pasan a las funciones subyacentes para personalizar la apariencia de los puntos y la línea de regresión. *height* es el tamaño de cada subgráfico (en pulgadas). *aspect* es la relación de aspecto de los subgráficos. *ci* Nivel de confianza para la línea de regresión. Puede ser *None* (predeterminado), '*sd*', '*boot*', o un valor numérico. *truncate* determina si se truncan las líneas de regresión fuera del rango de datos. Existen otros argumentos para personalizar la apariencia del gráfico, como *xlabel*, *ylabel*, *title*, *xlim* y *ylim*, entre otros.
10	*import pandas as pd* datos = *pd.DataFrame*({	Crea gráficos de dispersión o de otro tipo, histogramas en cada eje. x o y son los datos de las variables

N°	Función	Descripción
	'X': [1, 2, 3, 4, 5, 6, 7, 8, 9], 'Y': [3, 4, 4, 5, 6, 7, 8, 9, 10] }) *sns.jointplot*(x='X', y='Y', *data*=datos, *kind*='*scatter*', *color*='*blue*', *height*=6, *ratio*=5, *space*=0.2, *xlim*=(0, 10), *ylim*=(0, 12), *marginal_ticks*=*True*)  	que se desean visualizar. *data* es el *DataFrame* que contiene los datos. *kind* es el tipo de gráfico central. Puede ser '*scatter*' (predeterminado, dispersión), '*kde*' (distribuciones de densidades), '*hist*' (histogramas), '*hex*' (hexágonos), '*reg*' (regresión lineal), entre otros. *color* determina el color de los marcadores del gráfico de dispersión (solo para '*scatter*'). *height* es el tamaño del gráfico (en pulgadas). *ratio* es la relación de aspecto de la figura. *space* es el espacio adicional para las distribuciones marginales. *dropna* si es *True*, se excluyen los valores *NaN* al crear el gráfico. *xlim* y *ylim*: son los límites de los ejes X e Y. *marginal_ticks* si es *True*, agrega marcas en los ejes de las distribuciones marginales. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como *xlabel*, *ylabel* y *title*, entre otros.
11	*import pandas as pd* datos = *pd.DataFrame*({ 'Categoría': ['A', 'B', 'A', 'B', 'A'], 'X': [1, 2, 3, 4, 5],	Crea una matriz de gráficos de dispersión. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir

N°	Función	Descripción
	'Y': [3, 4, 4, 5, 6] }) *sns.pairplot*(*data*=datos, *hue*='Categoría', *palette*='*Set2*', *diag_kind*='*kde*', *kind*='*scatter*')  	los datos en categorías y mostrarlas en colores diferentes. *vars* es una lista con variables numéricas que se utilizan para crear los gráficos. Por defecto se utilizan todas las variables numéricas en el *DataFrame*. *diag_kind* determina el tipo de gráfico en la diagonal principal. Puede ser 'auto' (predeterminado), '*hist*', '*kde*', o una función personalizada. *kind* determina el tipo de gráficos fuera de la diagonal principal. Puede ser '*scatter*', '*reg*', '*kde*', '*hex*', o una función personalizada. *palette* es la paleta de colores que se utiliza para colorear las barras. *plot_kws* y *diag_kws* son un diccionario con los argumentos que se pasan a las funciones subyacentes para personalizar la apariencia de los gráficos. *height* es el tamaño de cada gráfico en la matriz (en pulgadas). *aspect* es la relación de aspecto de los gráficos. *corner* determina si se muestran los gráficos en la mitad superior derecha de la matriz cuando *hue* está presente. *corner_kws* es un diccionario de argumentos para personalizar los gráficos en la esquina.
12	*import pandas as pd* datos = *pd.DataFrame*({ 'Categoría': ['A', 'B', 'A', 'B', 'A'],	Crea varios tipos de gráficos categóricos, como gráficos de barras, gráficos de violín y gráficos de caja, entre otros. x o y son los datos de las variables

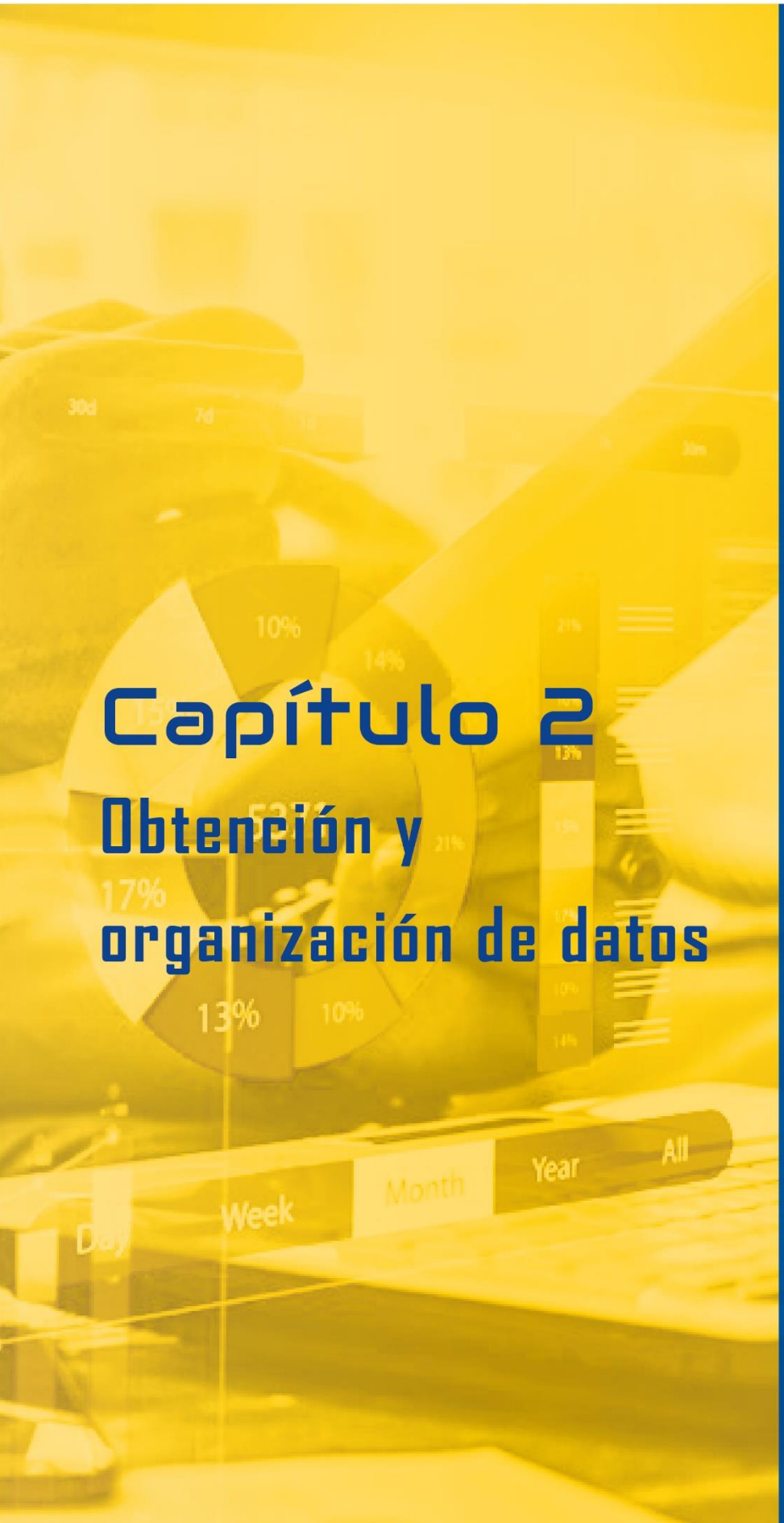
N°	Función	Descripción
	'Valor': [10, 15, 12, 17, 11] }) *sns.catplot*(x='Categoría', y='Valor', *data*=datos, *kind*='*bar*', *palette*='*Set2*', *height*=5, *aspect*=1.5)  	que se desean visualizar. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *kind* determina el tipo de gráficos fuera de la diagonal principal. Puede ser '*strip*' (gráfico de dispersión), '*swarm*' (gráfico de dispersión con ajuste), '*box*' (gráfico de caja), '*violin*' (gráfico de violín), '*bar*' (gráfico de barras), '*count*' (gráfico de conteo), entre otros. *palette* es la paleta de colores que se utiliza para colorear las barras. *height* es el tamaño de cada subgráfico (en pulgadas). *aspect* es la relación de aspecto de los subgráficos. *col* y *row* son variables categóricas opcionales que dividen los datos en múltiples subgráficos (columnas y filas) según estas variables *order*, *hue_order* es un vector con los nombres de las columnas del *DataFrame* que se va a representar en el orden en el que se desea las columnas. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como *xlabel*, *ylabel*, *title*, *xlim* y *ylim*, entre otros.

N°	Función	Descripción
13	*import pandas as pd* datos = *pd.DataFrame*({ 'X': [1, 2, 3, 4, 5], 'Y': [3, 3.5, 4.5, 5, 6], 'tipo': ['A', 'B', 'A', 'B', 'A'] }) *sns.relplot*(x='X', y='Y', *data*=datos, *hue*= *'tipo'*, *style*= *'tipo'*, *size*=None, *height*=3, *aspect*=1.3) 	Crea gráficos que muestran la relación entre dos variables numéricas. (*scatter plots*) x o y son los datos de las variables que se desean visualizar. *data* es el *DataFrame* que contiene los datos. *hue* es una variable categórica opcional que se utiliza para dividir los datos en categorías y mostrarlas en colores diferentes. *style* es una variable categórica opcional que se utiliza para cambiar el estilo de las marcas (Puede ser la misma utilizada en el parámetro *hue*). *size* es una variable numérica opcional que se utiliza para cambiar el tamaño de las marcas. *col* y *row* son variables categóricas opcionales que dividen los datos en múltiples subgráficos (columnas y filas) según estas variables. *palette* es la paleta de colores que se utiliza para colorear las barras. *height* es el tamaño de cada subgráfico (en pulgadas). *aspect* es la relación de aspecto de los subgráficos. *facet_kws* es un diccionario de argumentos para personalizar los subgráficos generados por las variables *col* y *row*. *markers* es una lista con los tipos de marcadores personalizados. *markersize* determina el tamaño de los marcadores. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una

N°	Función	Descripción
		sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como *xlabel*, *ylabel*, *title*, *xlim* y *ylim*, entre otros.
14	datos = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] *sns.heatmap*(*data*=datos, *annot*=*True*, *cmap*='*coolwarm*', *linewidths*=0.5, *linecolor*='*gray*', *vmin*=0, *vmax*=10, *cbar*=*True*) 	Genera mapas de calor. *data* es el *DataFrame* que contiene los datos. *annot* si es True, muestra los valores reales en las celdas del mapa de calor. *fmt* determina el formato de los números en las celdas cuando *annot=True*. *cmap* es la paleta de colores que se utiliza para codificar los valores en el mapa de calor. *linewidths* y *linecolor* determina el ancho y el color de las líneas entre las celdas del mapa de calor. *vmin* y *vmax* determinan los valores mínimos y máximos para la escala de colores. *cbar* si es *True*, muestra una barra de colores (*colorbar*) en el lado derecho del mapa de calor. *cbar_kws* es un diccionario de argumentos para personalizar la apariencia de la barra de colores. *xticklabels* y *yticklabels* determinan si se muestran las etiquetas de los ejes X e Y. *square* si es *True*, asegura que las celdas del mapa de calor sean cuadradas. *linewidths* y *linecolor*: determinan el ancho y el color de las líneas entre las celdas. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una

N°	Función	Descripción
		sola figura. Existen otros argumentos para personalizar la apariencia del gráfico, como *xlabel*, *ylabel*, *title*, entre otros.
15	*sns.set_palette*(["*#FF5733*", "*#33FF57*", "*#3366FF*"])	Permite personalizar la paleta de colores utilizada en los gráficos. El argumento es un vector con el número de colores igual al número de elementos categóricos de los datos.
16	*sns.despine*(*right*=*True*, *top*=*True*)	Elimina los márgenes (espina dorsal) de un gráfico. En el ejemplo se elimina los bordes superior y derecho del gráfico. *fig* indica la figura de *matplotlib* que se desea modificar. Si no se especifica, se utiliza la figura actual. *ax* es el eje en el que se dibuja el gráfico, cuando hay varios en una sola figura. *top*, *right*, *left*, *bottom* si es *True*, se elimina el eje indicado, (superior, derecha, izquierda o inferior del gráfico, respectivamente). *offset* es la distancia en puntos desde el borde del gráfico hasta el eje. Por defecto, 10 puntos. *trim* si es *True*, se recortan los ejes cuando se eliminan los bordes.

Nota. Los autores**.**



2

Capítulo 2

Obtención y organización de datos

Capítulo 2

2. Obtención y organización de datos

En la actualidad, más que nunca, los datos han emergido como una moneda invaluable que impulsan las decisiones informadas en diversos ámbitos. En el mantenimiento industrial es posible obtener gran cantidad de datos a través de los sistemas de gestión de mantenimiento asistido por ordenador (GMAO), por los sistemas de monitoreo de la condición o mediante los sistemas de control automatizados de las máquinas.

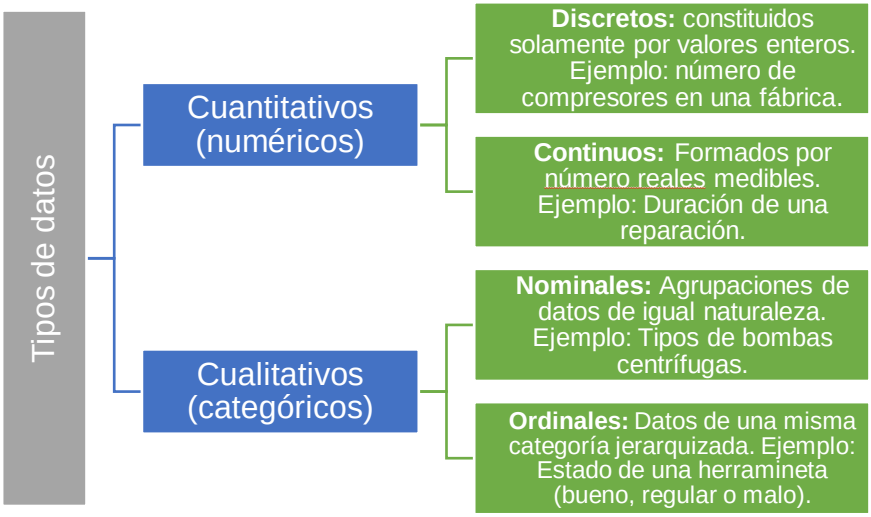
Estas cantidades grandes de datos por sí mismas son incapaces de transmitir información; por lo que es imperioso la aplicación de herramientas que permitan explorar estos datos y estructurarlos de tal forma que se facilite su manipulación y análisis posterior, lo que es crucial para generar conocimientos significativos y tomar decisiones fundamentadas.

2.1 Tipos de datos

De forma general los datos se clasifican en cuantitativos o numéricos y cualitativos o categóricos, mismos que se subdividen tal como se indica en la Figura 2.1.

Figura 2.1

Clasificación de los datos.



Nota. Los autores**.**

Los datos pueden contenerse en cualquier estructura de almacenamiento, de forma vectorial o matricial, en objetos como *list*, *array*, *lista*, *tupla*, *DataFrame* o *Series*, como se observa en los siguientes ejemplos donde se utilizan listas:

Listas con datos discretos:

```
# Número de compresores en 5 fábricas de inyección de plástico:
compresores = [4, 2, 1, 5, 3]
# Número de fallas anuales de 4 bombas centrífugas de 6 m3/h:
fallas = [10, 7, 9, 9]
# Unidades vendidas al mes de rodamientos 6204 de enero a junio
rodamientos = [425, 392, 405, 265, 305, 496]
```

Listas con datos continuos:

```
# Disponibilidad semanal de agosto de una máquina
disponibilidad = [0.954, 0.958, 0.950, 0.951]
# Mediciones de temperatura de un rodamiento
temperatura = [32.5, 34.1, 29.5, 30.6, 33.3]
# Mediciones de vibraciones de un ventilador radial en mm/s
vibracion = [5.2, 4.8, 3.5, 5.1, 4.8, 4.8, 3.9]
```

Listas con datos nominales:

```
# Marcas de rodamientos de un almacén
rodamientos = ['SKF', 'FAG', 'NSK', 'TIMKEN', 'KOYO']
# Electrodos revestidos más usados por el departamento de mantenimiento
de una empresa
electrodos = ['AWS 6011', 'AWS 6013', 'AWS E7018', 'AWS E316-L16']
# Elementos que más fallan en una bomba centrífuga
elementos = ['sello mecánico', 'acople', 'impulsor', 'rodamientos']
```

Listas con datos ordinales:

```
# Estado de un elemento susceptible a deterioro
estado = ['excelente', 'bueno', 'regular', 'malo']
# Límites de severidad del estado de un elemento
limite = ['alerta', 'alarma']
# Niveles de prioridad a las tareas de mantenimiento
nivel = ["alta", "media", "baja"]
```

2.2 Población y muestra

La **población** se refiere al conjunto completo de todos los elementos que cumplen ciertas características comunes en un estudio. Sin embargo, en la mayoría de los casos, es impracticable o costoso recopilar datos de toda la población. Aquí es donde entra en juego la **muestra**, que es un subconjunto seleccionado de la población (Figura 2.2). Si no se comprende la diferencia entre ambos, la muestra podría no ser representativa de la población en su totalidad, lo que podría conducir a conclusiones erróneas y decisiones incorrectas (Spiegel & Stephens, 2009, p. 1).

Figura 2.2

Población y muestra.

Nota. Adaptado de Lifeder (2020).

En la mayoría de los casos dentro del mantenimiento industrial, los datos son del tipo numérico continuo, puesto que son obtenidos mediante mediciones de variables técnicas como son las medidas de longitud, caudal, velocidad, temperatura,

presión, vibración, ruido, entre otros; y de variables de gestión como la disponibilidad, costos, tiempo entre fallas (TBF, del inglés *Time Between Failure*), tiempo hasta la recuperación (TTR, del inglés *Time To Restoration*) (UNE-EN 13306, 2018, p. 24), y así por el estilo.

Estas variables a su vez provienen de **poblaciones infinitas** puesto que sin importar la cantidad de mediciones que se haga, siempre va a ser posible añadir una más, sin que se logre obtener la última medida. En tanto que, en el caso de los datos discretos, nominales y ordinales, es común que se puedan conseguir el total de elementos, por lo que se le conoce como **población finita**. A pesar de eso en el caso de poblaciones finitas, pero de un gran número de elementos o que total sea desconocido, como el total de hojas de un árbol o el total de bombas centrífugas en buen estado de funcionamiento en el Ecuador, se les consideran como poblaciones infinitas.

Cuando se trabaja con poblaciones infinitas o resulta muy costosos o impráctico trabajar con poblaciones finitas, se puede extraer una muestra de "n" elementos, lo suficientemente grande para considerarse representativa de la población. En algunos

libros se indica que por convención y sin exponer argumento alguno, una muestra grande tiene 30 o más elementos ($n \geq 30$); por otro lado, la misma bibliografía expone las ecuaciones con su respectiva demostración para el cálculo del tamaño de la muestra, mismas que se indican a continuación y cuyo resultado siempre se debe redondear hacia el primer entero superior (Triola, 2018, pp. 307-332).

Tamaño de la muestra n para estimar la proporción p de poblaciones infinitas:

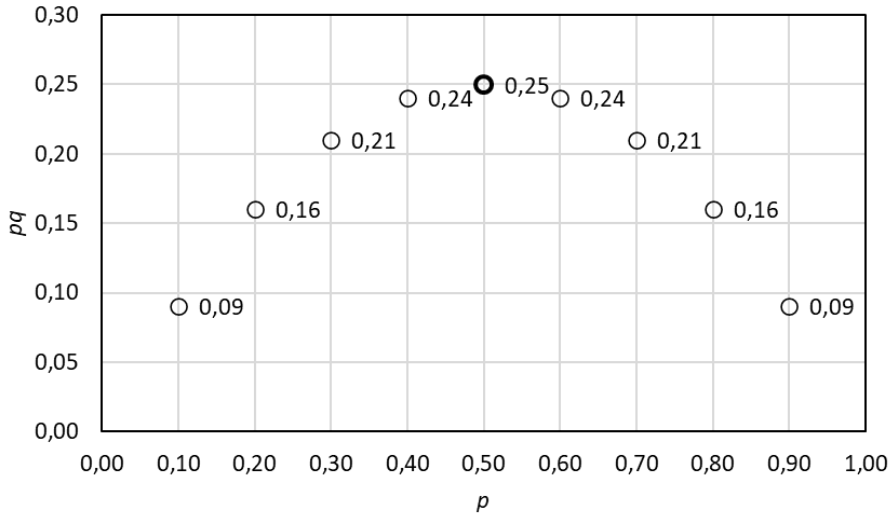
$$n = \frac{z_{\alpha/2}^2 \cdot p \cdot q}{E^2}, \tag{1}$$

done $z_{\alpha/2}$ es la puntuación z crítica basada en el nivel de confianza deseado donde el más usado es de $1{,}960$ correspondiente a una confianza de $0{,}95\ (o\ 95\%)$ y una significancia de $0{,}05$ que es igual a $z_{\alpha/2}(0{,}95) = z\big((0{,}95 + 1)/2\big)$, p es la proporción muestral de un estudio inicial de x éxitos en una muestra de tamaño n_i ($p = x/n_i$), q es la proporción muestral de fracasos en una muestra de tamaño n_i ($q = 1 - p$), y E es el margen de error deseado como índice.

Cuando p y q son desconocidos se le puede dar el valor de 0,50 cuyo producto pq es igual a 0,25 que es el máximo posible como se puede ver en la Figura 2.3, dando a su vez como resultado un tamaño de muestra mayor que si se conociera p y fuera diferente de 0,50.

Figura 2.3

Valor máximo de pq para p desconocido



Nota. Los autores**.**

Ejemplo:

En un lote de tamaño desconocido de pernos M10 (diámetro 10 mm), existen dos diferentes longitudes 30 mm y 50 mm, en una muestra inicial de 20 pernos se encontraron 3 de

longitud 30 mm (considerados como aciertos) por lo que $p = 0{,}15$. Calcular el tamaño de la muestra para estimar la proporción con un error de 5% y una confianza del 95%.

Código:

```
# Importación de las librerías numpy y scipy.stats
import numpy as np
import scipy.stats as stats

# Datos
confianza = 0.95
p = 0.15
E = 0.05
q = 1-p
z = stats.norm.ppf((confianza+1)/2) # z de alpha medios

# Cálculo de la muestra
n = z**2*p*q/ E**2 # Ecuación (1)
n = np.ceil(n) # primer entero superior
print('El tamaño de la muestra es:',n)
```

Salida:

El tamaño de la muestra es: 196

Tamaño de la muestra n para estimar la proporción p de poblaciones finitas:

$$n = \frac{z_{\alpha/2}^2 \cdot N \cdot p \cdot q}{p \cdot q \cdot z_{\alpha/2}^2 + (N-1) \cdot E^2}, \qquad (2)$$

donde N es el tamaño de la población.

Ejemplo:

Calcular el tamaño de la muestra para estimar la proporción en el ejercicio anterior, si el tamaño de la población es 500. Hay que considerar que las librerías ya se importaron al igual que ya se dieron valores a algunas variables, por lo que ya no es necesario hacerlo nuevamente.

Código:

```
N = 500
n = z**2*N*p*q/(z**2*p*q+(N-1)*E**2) # Ecuación (2)
n = np.ceil(n) # primer entero superior
print('El tamaño de la muestra es:',n)
```

Salida:

El tamaño de la muestra es: 141

Tamaño de la muestra n para estimar la media μ de poblaciones infinitas y σ conocida:

$$n = \frac{z_{\alpha/2}^2 \cdot \sigma^2}{E^2}, \qquad (3)$$

donde σ es la desviación estándar poblacional que tiene la misma unidad que los elementos de la población y de igual marera, E también tiene la misma unidad que los elementos de la población.

Ejemplo:

En un lote de tamaño desconocido de bombas centrífugas, se ha realizado la medición del caudal arrojando como resultado una desviación estándar poblacional de $\sigma = 16{,}44\ l/s$. Calcular el tamaño de la muestra para estimar la media con un error de $5\ l/s$ y una confianza del 95%.

Código:

```
sigma = 16.44
E = 5
n = (z*sigma/E)**2 # Ecuación (3)
n = np.ceil(n) # primer entero superior
print('El tamaño de la muestra es:',n)
```

Salida:

El tamaño de la muestra es: 42

Tamaño de la muestra n para estimar la media μ de poblaciones finitas y σ conocida:

$$n = \frac{z_{\alpha/2}^2 \cdot N \cdot \sigma^2}{\sigma^2 \cdot z_{\alpha/2}^2 + (N-1) \cdot E^2}, \qquad (4)$$

Ejemplo:

Calcular el tamaño de la muestra para estimar la media en el ejercicio anterior, si el tamaño de la población es 5000.

```
N = 5000
n = z**2*N*sigma**2/(z**2*sigma**2+(N-1)*E**2)  # Ecuación (4)
n = np.ceil(n) # primer entero superior
print('El tamaño de la muestra es:',n)
```

Salida:

El tamaño de la muestra es: 42

Tamaño de la muestra n para estimar la media μ de poblaciones infinitas y σ desconocida:

$$n = \frac{t_{\alpha/2}^2 \cdot s^2}{E^2}, \tag{5}$$

done $t_{\alpha/2}$ es la puntuación t crítica basada en el nivel de confianza deseado donde el más usado correspondiente a una confianza de $0{,}95\ (o\ 95\%)$ y una significancia de $0{,}05$ que es igual a $t_{\alpha/2}(0{,}95; gl) = t\big((0{,}95+1)/2; gl\big)$, gl son los grados de libertad igual a $gl = n_i - 1$, n_i es el tamaño de una muestra inicial y s es la desviación estándar muestral.

Ejemplo:

En un lote de tamaño desconocido de bombas centrífugas, se ha tomado una muestra inicial de 20 mediciones del caudal en l/s, arrojando como resultado una $s = 16{,}44\ l/s$. Calcular el tamaño de la muestra para estimar la media con un $E = 5\ l/s$ y una confianza del 95%.

Código:

```
ni = 20
gl = ni-1 # grados de libertad
s = 16.44
t = stats.t.ppf((confianza+1)/2, df=gl) # t de alpha medios

n = (t*s/E)**2 # Ecuación (5)
n = np.ceil(n) # primer entero superior
print('El tamaño de la muestra es:',n)
```

Salida:

El tamaño de la muestra es: 48

Ejemplo:

Calcular el tamaño de la muestra para estimar la media de la población infinita y σ desconocida del ejercicio anterior con una confianza del 95%, partiendo de una muestra inicial de 20 datos e indicando la importación de librerías.

Código:

```
# Importación de las librerías numpy y scipy.stats
import numpy as np
import scipy.stats as stats

caudal = [366.2, 379.1, 383.2, 395.9, 371.1, 347.0, 361.0, 361.9, 351.3, 350.5,
378.4, 383.5, 360.9, 349.1, 326.0, 349.6, 350.5, 358.1, 364.1, 376.1]
confianza = 0.95
ni = len(caudal)
s = np.std(caudal, ddof=1)
t = stats.t.ppf((confianza+1)/2, df=ni-1) # t de alpha medios
n = (t*s/E)**2 # Ecuación (5)
n = np.ceil(n) # primer entero superior
print('El tamaño de la muestra es:',n)
```

Salida:

El tamaño de la muestra es: 48

2.3 Métodos para recolectar datos

En la recolección y toma de datos se debe tomar en cuenta las siguientes consideraciones:

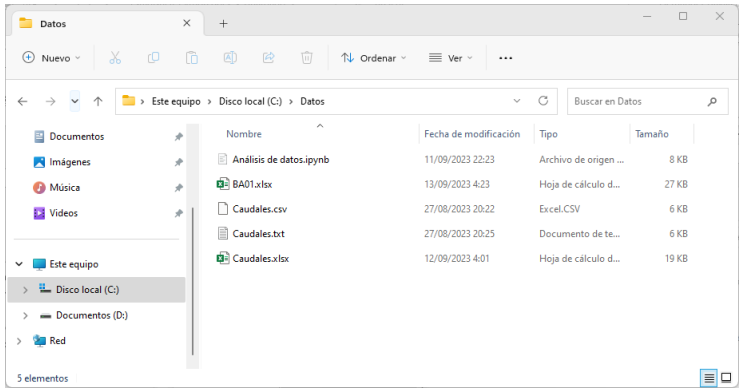
1. Los datos deben pertenecer a una misma variable.
2. Deben ser lo más exacto posible.
3. Deben estar tomados en condiciones normales.
4. Deben ser tomados por personas responsables.

Los datos pueden recopilarse en archivos como hojas de cálculo o bloque de notas, en el momento del registro o como producto de la exportación de información de un sistema informático de gestión de mantenimiento como una GMAO o de un sistema informático de gestión empresarial como una ERP. Posteriormente estos datos pueden importarse a *Python* como se indica a continuación, no sin antes importar todas las librerías que se requieren en el ejemplo que se desarrolla de aquí en adelante.

El primer paso es crear un archivo de *Python Jupyter* (**.ipynb*) con el nombre y en un directorio que elija el lector, en el cual también se deben guardar los archivos con los datos (Figura 2.4) que se puede descargar del siguiente enlace: https://acortar.link/vEeWcM.

Figura 2.4

Ubicación del archivo de Python Jupyter y de los que se van a importar los datos.



Nota. Los autores**.**

Para importar los datos a Python desde archivos como xlsx, csv o txt, se puede utilizar la librería Pandas como se indica a continuación (Valverde et al., 2023, pp. 217-224).

Código:

Importación de librerías:

```
import numpy as np
import pandas as pd
```

Importación de un archivo Excel:

```
# Importación la Hoja1 de un archivo de Excel
libro = pd.ExcelFile('Caudales.xlsx')

# Consulta de los nombres de las Hojas del archivo de Excel
print(libro.sheet_names)
# Asignación de la Hoja1 a tabla_caudales como un DataFrame
tabla_caudales = libro.parse('Hoja1')
# Cierre del archivo de Excel Caudales.xlsx
libro.close()
```

Salida:

['Hoja1', 'Hoja2', 'Hoja3']

Importación de un archivo csv:

```
# Si la "," es el separador entre datos y el "." el separador de decimales
tabla_caudales = pd.read_csv('Caudales.csv')
# Si el ";" es el separador entre datos y la "," el separador de decimales
tabla_caudales = pd.read_csv('Caudales.csv', decimal=',', sep=';')
```

Importación de un archivo txt:

```
# Si los datos tienen encabezado
tabla_caudales = pd.read_csv(' Caudales.txt', sep='\t')
# Si los datos tienen encabezado y la "," como separador de decimales
tabla_caudales = pd.read_csv('Caudales.txt', sep='\t', decimal=',')
# Si los datos no tienen encabezado
tabla_caudales = pd.read_csv(' Caudales.txt', sep='\t', header=None)
```

Una vez importados los datos, se debe hacer una exploración rápida de estos mediante las funciones *info*(), *head*() o *tail*() de *Pandas* (Tabla 1.2).

Código:

```
# Eliminación de las filas con valores nulos
tabla_caudales = tabla_caudales.dropna()
# Mostrar la información general de tabla_caudales
print(tabla_caudales.info())
# Mostrar las primeras filas, por defecto muestra las 5 primeras filas
print(tabla_caudales.head())
```

Salida:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   n       500 non-null    int64
 1   caudal  500 non-null    float64
dtypes: float64(1), int64(1)


memory usage: 7.9 KB
None
   n  caudal
0  1   366.2
1  2   379.1
2  3   383.2
3  4   395.9
4  5   371.1
```

La información de salida indica que el archivo importado con el nombre "libro" tiene 3 hojas de cálculo de nombres "Hoja1", "Hoja2" y "Hoja3", donde la "Hoja1" se le asignó en nombre de "tabla_caudales" mismo que es un *DataFrame* (tabla) de 2 columnas y 500 filas o datos, con valores no nulos de índices que van desde el 0 al 499, donde la primera columna tiene valores enteros y la segunda columna tiene valores flotantes o decimales.

Con la función *value_counts*() de *Pandas*, se puede observar la frecuencia de los elementos individuales de la muestra o de la población, con el siguiente código:

Código:

```
# Asignación de la variable caudal para la columna de datos de los caudales
caudal = tabla_caudales['caudal']
# Frecuencia de los valores individuales
print(pd.value_counts(caudal))
```

Salida:

```
377.5    5
363.2    5
369.9    5
375.2    4
367.8    4
        ..
368.3    1
371.5    1
349.9    1
375.0    1
372.5    1
Name: caudal, Length: 325, dtype: int64
```

2.4 Tabla de frecuencias

Para crear la tabla de frecuencias de los datos de caudales, primeramente, se debe determinar el número de clases, para lo cual se prefiere utilizar la ley de Sturges ($q = 1 + log_2 500 \approx 10$) que de preferencia debe ser un número impar (Triola, 2018, p. 51), posteriormente se puede hacer variaciones de este valor con miras a que el histograma se vea coherente (Figuras 7.1 y 7.2). El logaritmo de base 2 puede calcularse con la función *log2*(n) de *NumPy* (Tabla 1.1).

Código:

```
# Tamaño de la muestra
n = len(caudal)
# Cálculo del rango
rango = caudal.max() - caudal.min()
# Calculo del número de clases. Se recomienda que sea impar
q = round(1 + np.log2(n)) # en este caso q = 10
# Cambio de q a 9 para ajustar, caso contrario deje q = q
q = 9
# Cálculo del ancho de clase
a = rango/q
# Cambio del ancho de clase a 11,1 para ajustar, caso contrario deje a = a
a = 11.1
# Valor mínimo de la muestra
x_min = caudal.min()
# Cambio del mínimo de la muestra para ajustar, caso contrario deje x_min
```

```
= x_min
x_min = x_min

print('Tamaño de la muestra:', n)
print('Rango:              ', rango)
print('Número de clases:   ', q)
print('Ancho de clases:    ', a)
```

Salida:

```
Tamaño de la muestra: 500
Rango:                99.59999999999997
Número de clases:     9
Ancho de clases:      11.1
```

Una vez calculados los parámetros iniciales, se procede a construcción de la tabla de frecuencia dentro de un objeto *DataFrame*.

Código:

```
# Creación de un DataFrame vacío con el nombre tf
tf = pd.DataFrame()
# Creación de la columna 'Clases' con los números de clases
tf['Clase'] = list(range(1, q + 1))

# Creación de la columna 'Linf' vacía
tf['Linf'] = np.full(shape=q, fill_value=np.nan)
# Recurrencia para llenar con los límites inferiores
for i in range(q):
```

```
    tf.loc[i, 'Linf'] = round(x_min + i * a, 1)

# Creación de la columna 'Lsup' con los límites superiores
tf['Lsup'] = round(tf['Linf'] + a, 1)

# Creación de la columna 'x' con las marcas de clase
tf['x'] = (tf['Lsup'] + tf['Linf']) / 2

# Creación de la columna 'f' vacía
tf['f'] = np.full(shape=q, fill_value=np.nan)
# Recurrencia para llenar con la frecuencia de cada clase
for i in range(q):
    k = 0
    if i == 0:
        for j in range(n):
            if caudal[j] <= tf['Lsup'][i]:
                k += 1
        tf.loc[i, 'f'] = k
    else:
        for j in range(n):
            if caudal[j] > tf['Linf'][i] and caudal[j] <= tf['Lsup'][i]: # cerrardo [Linf
- Lsup) abierto
                k += 1
        tf.loc[i, 'f'] = k

# Cálculo de la frecuencias acumuladas
tf['Fa'] = tf['f'].cumsum()

# Cálculo de la frecuencias relativas
tf['fr'] = round(tf['f'] / n, 4)

# Cálculo de la frecuencias relativas acumuladas
tf['Fra'] = tf['fr'].cumsum()

# Impresión de la tabla de frecuencias
print(tf)
```

Salida:

	Clase	Linf	Lsup	x	f	Fa	fr	Fra
0	1	318.1	329.2	323.65	2.0	2.0	0.004	0.004
1	2	329.2	340.3	334.75	11.0	13.0	0.022	0.026
2	3	340.3	351.4	345.85	52.0	65.0	0.104	0.130
3	4	351.4	362.5	356.95	106.0	171.0	0.212	0.342
4	5	362.5	373.6	368.05	131.0	302.0	0.262	0.604
5	6	373.6	384.7	379.15	108.0	410.0	0.216	0.820
6	7	384.7	395.8	390.25	62.0	472.0	0.124	0.944
7	8	395.8	406.9	401.35	22.0	494.0	0.044	0.988
8	9	406.9	418.0	412.45	6.0	500.0	0.012	1.000

Capítulo 3

Medidas de tendencia central

Capítulo 3

3. Medidas de tendencia central

Para el cálculo de las medidas de tendencia central, se utiliza las funciones de las librerías *NumPy* y *Pandas*, aplicadas a los datos importados a *Python Jupyter* en el Capítulo 2.

Importación de librerías:

```
import numpy as np
import pandas as pd
```

Las medidas de tendencia central ampliamente utilizadas en la estadística descriptiva son la media, la moda y mediana.

3.1 Media

3.1.1 Media aritmética

La media aritmética de los valores independientes de una muestra $\bar{x}$ se calcula con la Ecuación (6) donde se suman los elementos de la muestra desde el primero hasta el último y se divida para el tamaño de la muestra.

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}, \tag{6}$$

donde x_i es el i ésimo elemento de la muestra y n es el tamaño de la muestra.

Para calcular media de los valores independientes de la población μ, se emplea la siguiente ecuación:

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N}, \tag{7}$$

donde x_i es el i ésimo elemento de la población y N es el tamaño de la población.

En *Python* se puede calcular de 2 maneras:

Código:

Opción 1:

```
n = len(caudal)
media = caudal.sum()/n
print('%.4f' % media)
```

Salida:

370.0364

Opción 2:

```
media = caudal.mean()
print('%.4f' % media)
```

Salida:

370.0364

La media aritmética de los valores agrupados $\bar{x}$ puede calcularse con la siguiente ecuación:

$$\bar{x} = \frac{\sum_{i=1}^{q} x_i \cdot f_i}{n} = \sum_{i=1}^{q} x_i \cdot fr_i \,, \tag{8}$$

donde x_i es la i ésima marca de clase, f_i es la i ésima frecuencia absoluta, fr_i es la i ésima frecuencia relativa, q es el número de clases y n es el tamaño de la muestra.

Código:

```
media = (tf['x']*tf["fr"]).sum()
print('%.4f' % media)
```

Salida:

369.6262

3.1.2 Media abreviada

La media abreviada de los valores independientes se calcula con la siguiente ecuación:

$$\bar{x} = A + \frac{\sum_{i=1}^{n} d_i}{n}, \tag{9}$$

donde A es un número cualquiera, d_i es la i ésima diferencia entre $x_i - A$.

Código:

```
A = 100
n = len(caudal)
media = A + (caudal-A).sum()/n
print('%.4f' % media)
```

Salida:

370.0364

La media abreviada de los valores agrupados se calcula con la siguiente ecuación:

$$\bar{x} = A + \sum_{i=1}^{q} fr_i \cdot d_i \,, \tag{10}$$

Código:

```
A = 100
media = A + (tf['fr']*(tf['x']-A)).sum()
```

Salida:

369.6262

3.1.3 Media geométrica

La media geométrica de los valores independientes G se calcula con la siguiente ecuación:

$$G = \sqrt[n]{\prod_{i=1}^{n} x_i} \,, \tag{11}$$

Código:

```
import scipy.stats as stats
G = stats.gmean(caudal)
print('%.4f' % G)
```

Salida:

369.6882

La media geométrica de los valores agrupados G se calcula con la siguiente ecuación:

$$G = 10^{\sum_{i=1}^{q} fr_i \cdot log_{10} x_i}, \quad (12)$$

Código:

```
import numpy as np
G = pow(10, (tf['fr']*np.log10(tf['x'])).sum())
print('%.4f' % G)
```

Salida:

369.2649

3.1.4 Media armónica

La media armónica de los valores independientes se calcula con la siguiente ecuación:

$$H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}, \tag{13}$$

Código:

```
n = len(caudal)
H = n/(1/caudal).sum()
print('%.4f' % H)
```

Salida:

369.3403

La media armónica de los valores agrupados se calcula con la siguiente ecuación:

$$H = \frac{n}{\sum_{i=1}^{q} \frac{f_i}{x_i}}, \tag{14}$$

Código:

```
n = len(caudal)
H = n/(tf['f']/tf['x']).sum()
print('%.4f' % H)
```

Salida:

368.9038

3.1.5 Media cuadrática

Conocida también como Raíz Cuadrada Media (RCM) o más comúnmente en aplicaciones físicas como RMS, del inglés *Root Mean Square*, y se define como la raíz cuadrada del promedio de los cuadrados (Spiegel & Stephens, 2009, p. 66; Triola, 2018, p. 96), esto es:

$$RMS = \sqrt{\overline{x_i{}^2}}\,, \tag{15}$$

Código:

```
rms = ((caudal**2).mean())**(1/2)
print('%.4f' % rms)
```

Salida:

370.3849

O como se indica con la siguiente Ecuación:

$$RMS = \sqrt{\frac{\sum_{i=1}^{n} {x_i}^2}{n}}, \tag{16}$$

Código:

```
n = len(caudal)
rms = ((caudal**2).sum()/n)**(1/2)
print('%.4f' % rms)
```

Salida:

370.3849

Para calcular el RMS de los valores agrupados, se emplea la siguiente ecuación:

$$RMS = \sqrt{\frac{\sum_{i=1}^{q} f_i \cdot {x_i}^2}{n}} = \sqrt{\sum_{i=1}^{q} fr_i \cdot {x_i}^2}, \tag{17}$$

Código:

```
rms = ((tf['fr']*tf['x']**2).sum())**(1/2)
print('%.4f' % rms)
```

Salida:

369.9877

3.2 Moda

Es el elemento que más se repita en la muestra, pudiendo haber más de uno, en este caso se le denomina: muestra multimodal. No está sujeto al cálculo para los valores independientes; sin embargo, *Pandas* cuenta con la función *mode()* para encontrarlos.

Código:

```
moda = caudal.mode()
print(moda)
```

Salida:

```
0    363.2
1    369.9
2    377.5
```

Name: caudal, dtype: float64

La moda de los valores agrupados se calcula con la siguiente Ecuación:

$$Mo = L_{inf_i} + \left(\frac{f_i - f_{i-1}}{f_i - f_{i-1} + f_i - f_{i+1}}\right) \cdot a\,, \tag{18}$$

donde L_{inf_i} es el límite inferior de la tabla de frecuencias donde se encuentra la moda, f_i es la frecuencia absoluta donde se encuentra la moda, f_{i-1} es la frecuencia absoluta de la anterior clase donde se encuentra la moda, y f_{i+1} es la frecuencia absoluta de la siguiente clase donde se encuentra la moda.

Código:

```
# Búsqueda del índice de la clase donde está la moda
k = tf['f'].tolist().index(max(tf['f']))
# Cálculo de la moda
moda = tf['Linf'][k]+(tf['f'][k]-tf['f'][k-1])/(2*tf['f'][k]-tf['f'][k-1]-tf['f'][k+1])*a
print('%.1f' % moda)
```

Salida:

368.3

3.3 Mediana

La mediana de una muestra es el valor que separa los datos ordenados en dos partes iguales, donde la mitad de los datos (50%) están por debajo de la mediana y la otra mitad (50%) está por encima.

Para encontrar la mediana en una muestra se debe en primer lugar ordenar los datos de menor a mayor, y posteriormente se determina la posición de la mediana. Si el tamaño de la muestra es impar, la mediana es el valor en la posición central. Si el tamaño de la muestra es par, la mediana es el promedio de los dos valores centrales.

La mediana de los valores independientes Me de tamaño de muestra ***impar*** se calcula con la siguiente ecuación:

$$Me = \text{Dato en la posición } (n + 1) / 2\,, \tag{19}$$

La mediana de los valores independientes Me de tamaño de muestra ***par*** se calcula con la siguiente Ecuación:

$$Me = (Dato\ en\ la\ posición\ n / 2) + (Dato\ en\ la\ posición\ (n / 2) + 1) / 2\ , \quad (20)$$

Código:

Opción 1:

```
n = len(caudal)
if n % 2:
    mediana = sorted(caudal)[round(0.5*(n-1))]
else:
    x_ord, index = sorted(caudal), round(0.5 * n)
    mediana = 0.5 * (x_ord[index-1] + x_ord[index])
print(mediana)
```

Salida:

369.9

Opción 2:

```
import numpy as np
mediana = np.median(caudal)
print(mediana)
```

Salida:

369.9

La mediana de los valores agrupados Me se calcula con la siguiente Ecuación:

$$Me = L_{inf_i} + \frac{n * 0{,}50 - F_{i-1}}{f_i} \cdot a\,, \tag{21}$$

donde L_{inf_i} es el límite inferior de la tabla de frecuencias donde se encuentra la mediana, F_{i-1} es la frecuencia absoluta acumulada de la clase anterior de donde se encuentra la mediana, f_i es la frecuencia absoluta donde se encuentra la mediana y a es el ancho de clases.

Código:

```
# Búsqueda del índice de la clase donde está la mediana
k = min([i for i, valor in enumerate(tf['Fra']) if valor >= 0.5])
# Cálculo de la mediana
mediana = tf['Linf'][k]+(n*0.50-tf['Fa'][k-1])/tf['f'][k]*a
print('%.1f' % mediana)
```

Salida:

369.2

3.4 Ecuaciones de las medidas de tendencia central en *Markdown*

En las celdas de *Jupyter Notebook* es posible hacer ecuaciones mediante el empleo de código en *Markdown*, de forma similar a como se hace en *LaTex*. En la Tabla 3.1 se presenta el código para poder elaborar las Ecuaciones desde la (6) a la (21).

Tabla 3.1

Código en Markdown para elaborar las ecuaciones de las medidas de tendencia central.

Nombre	Código	Salida
Media aritmética de los valores independientes.	`$$\bar x = \frac{\sum_{i=1}^{n}{x_i}}{n}`	$\bar{x} = \frac{\sum_1^n x_i}{n}$
Media aritmética de los valores agrupados.	`$$\bar x = \sum_{i=1}^{q}{x_i*fr_i}`	$\bar{x} = \sum_{i=1}^{q} x_i * fr_i$
Media abreviada de los valores independientes.	`$$\bar x = A+ \frac{\sum_{i=1}^{n}{d_i}}{n}$$`	$\bar{x} = A + \frac{\sum_{i=1}^{n} d_i}{n}$

Nombre	Código	Salida
Media abreviada de los valores agrupados.	$$\bar x = A+ \sum_{i=1}^{q}{f_i*d_i}$$	$$\bar{x} = A + \sum_{i=1}^{q} fr_i * d_i$$
Media geométrica de los valores independientes.	$$ G = \sqrt[n]{\prod_{i=1}^{n} x_i}$$	$$G = \sqrt[n]{\prod_{i=1}^{n} x_i}$$
Media geométrica de los valores agrupados.	$$ G = 10^{\sum{f_i*log_{10}x_i}}$$	$$G = 10^{\sum_{i=1}^{q} fr_i * log_{10} x_i}$$
Media armónica de los valores independientes.	$$ H =\frac{n}{\sum_{i=1}^{n}{\frac{1}{x_i}}}$$	$$H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$
Media armónica de los valores agrupados.	$$ H =\frac{n}{\sum_{i=1}^{q}{\frac{f_i}{x_i}}}$$	$$H = \frac{n}{\sum_{i=1}^{q} \frac{f_i}{x_i}}$$
Media cuadrática de los valores independientes.	$$ RMS = \sqrt[]{\bar {x_i^{2}}}$$	$$RMS = \sqrt{\overline{x_i^2}}$$
Media cuadrática de los valores independientes.	$$ RMS = \sqrt[]{\frac{\sum_{i=1}^{n}{ x_i^{2}}}{n}}$$	$$RMS = \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}$$

Nombre	Código	Salida
Media cuadrática de los valores agrupados.	$$ RMS = \sqrt[]{\frac{\sum_{i=1}^{q}{fr_i*x_i^{2}}}{n}}$$	$RMS = \sqrt{\sum_{i=1}^{q} fr_i * {x_i}^2}$
Moda de los valores agrupados.	$$ Mo = {L_{inf}}_i+ \left(\frac{f_i-f_{i-1}}{f_i-f_{i-1}+f_i-f_{i+1}}\right)*a $$	$Mo = L_{inf_i} + \left(\frac{f_i - f_{i-1}}{f_i - f_{i-1} + f_i - f_{i+1}}\right) * a$
Mediana de los valores independientes.	$$ Me =\frac{Dato en la posición (n + 1)}{2}$$	$Me = \frac{Dato\ en\ la\ posición\ (n}{2}$
Mediana de los valores agrupados.	$$ Me = {L_{inf}}_i+\frac{n*0,50-F_{i-1}}{fi}*a$$	$Me = L_{inf_i} + \frac{n * 0{,}50 - F_{i-1}}{f_i} * a$

Nota. Los autores**.**

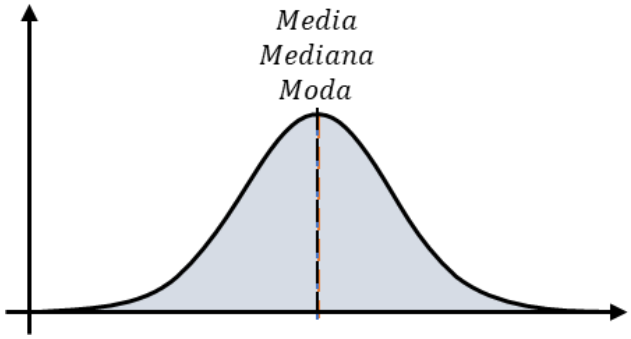
3.5 Análisis de las medidas de tendencia central

Cuando se analizan en conjunto las medidas de tendencia central como la media, mediana y moda, se puede obtener información valiosa sobre cómo se agrupan los valores alrededor de un punto central, con lo que se tiene más claridad sobre la forma de la distribución, teniendo 3 posibles casos (Lind et al., 2012, p. 69) distribución simétrica, distribución sesgada hacia la derecha y distribución sesgada hacia la izquierda.

Se considera distribución simétrica cuando las medidas de tendencia central son aproximadamente iguales como lo indica la siguiente Figura:

Figura 3.1

Distribución simétrica.

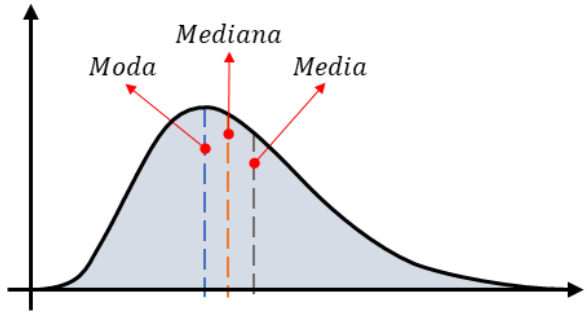


Nota. Adaptado de Kelmansky, 2009, p. 149.

En la distribución sesgada hacia la derecha, la media está desplazada hacia valores más altos, en tanto que la moda ocupa la ubicación donde la campana de Gauss es máxima, como se indica en la siguiente Figura:

Figura 3.2

Distribución sesgada hacia la derecha.

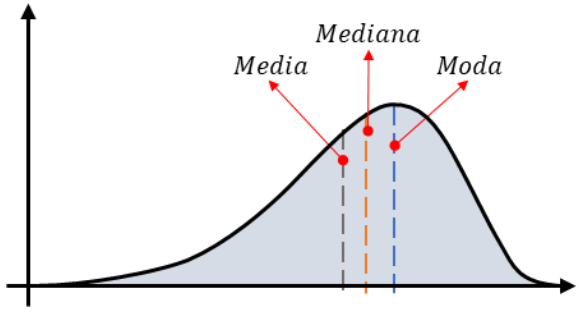


Nota. Adaptado de Spiegel & Stephens, 2009, p. 65.

Por último, en la distribución sesgada hacia la izquierda, la media está desplazada hacia valores más bajos, mientras que la moda nuevamente ocupa la ubicación donde la campana de Gauss es máxima, como se indica en la siguiente Figura:

Figura 3.3

Distribución sesgada hacia la izquierda.



Nota. Adaptado de Spiegel & Stephens, 2009, p. 65.

Como comentario general se puede observar que en las Figuras 3.2 y 3.3, la mediana se desplaza menos que la media, por lo que se deduce que es poco afectada por el sesgamiento de los datos, y es por lo que esta medida de tendencia central es la más representativa de una muestra.

Capítulo 4

Medidas de dispersión y posición

Capítulo 4

4. Medidas de dispersión y posición

Para el cálculo de las medidas de dispersión y posición, se utiliza las funciones de las librerías *NumPy* y *Pandas*, aplicadas a los datos importados a *Python Jupyter* en el Capítulo 2.

Importación de librerías:

```
import numpy as np
import pandas as pd
```

4.1 Rango

Es la diferencia entre el valor más alto de la muestra y el más bajo. Se calcula con la siguiente Ecuación:

$$rango = máximo - mínimo. \quad (22)$$

Código:

```
# Cálculo del rango
rango = caudal.max() - caudal.min()
print('%.1f' % rango)
```

Salida:

99.6

4.2 Desviación estándar

Es una medida de dispersión que indica la cantidad promedio por la cual los valores en una población o muestra de datos se alejan de la media.

Para calcular la desviación estándar poblacional de los valores independientes σ, se emplea la siguiente Ecuación:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}, \tag{23}$$

Código:

Opción 1:

```
# Tamaño de la población
N = len(caudal)
# Cálculo de la desviación estándar poblacional
des = (((caudal - caudal.mean())**2).sum()/N)**(1/2)
print('%.4f' % des)
```

Salida:

16.0642

Opción 2:

```
# Cálculo de la desviación estándar poblacional
des = caudal.std(ddof=0)
print('%.4f' % des)
```

Salida:

16.0642

Para calcular la desviación estándar muestral de los valores independientes s, se emplea la siguiente Ecuación:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}, \tag{24}$$

Código:

Opción 1:

```
# Tamaño de la muestra
n = len(caudal)
# Cálculo de la desviación estándar muestral
des = (((caudal - caudal.mean())**2).sum()/(n-1))**(1/2)
print('%.4f' % des)
```

Salida:

16.0803

Opción 2:

```
# Cálculo de la desviación estándar muestral
des = caudal.std(ddof=1)
print('%.4f' % des)
```

Salida:

16.0803

En el caso de los valores agrupados, se consideran dos casos; cuando $n \leq 30$ y cuando $n > 30$.

Para calcular la desviación estándar de los valores agrupados s de $n \leq 30$, se utiliza la siguiente Ecuación:

$$s = \sqrt{\frac{\sum_{i=1}^{q} f_i \cdot (x_i - \bar{x})^2}{n-1}}, \tag{25}$$

Código:

```
# Tamaño de la muestra
n = len(caudal)
# Cálculo de la desviación estándar muestral
des = ((tf['f']*(tf['x'] - (tf['f']*tf['x']).sum())**2).sum()/(n-1))**(1/2)
print('%.4f' % des)
```

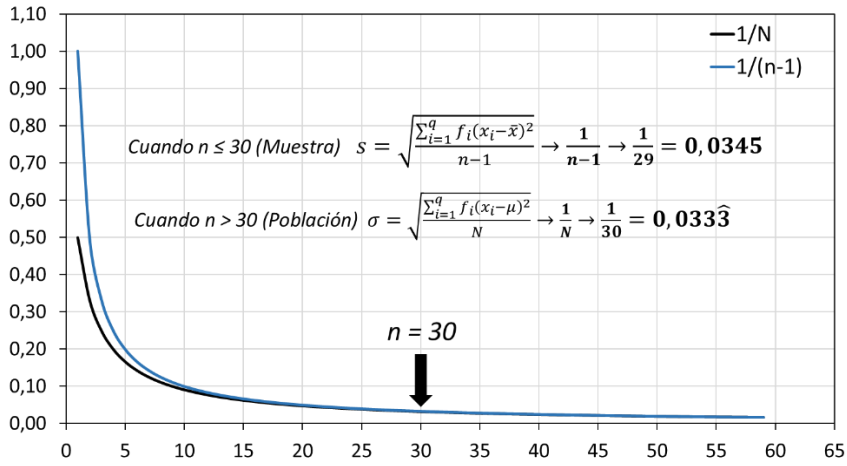
Salida:

16.3671

Si $n = 30$, la diferencia entre $1/(n-1)$ y $1/n$ es $0{,}0345 - 0.0333 = 0{,}0012$, y mientras más grande es la muestra, esta diferencia es cada vez menor, tal como se indica en la siguiente Figura:

Figura 4.1

Efecto de $\frac{1}{n-1}$ y de $\frac{1}{n}$ sobre el cálculo de la desviación estándar.



Nota. Los autores**.**

Esta observación se corrobora ya que el $\lim_{n\to\infty} \frac{1}{(n-1)} - \frac{1}{n} = 0$; por lo tanto, para muestras grandes la desviación estándar de valores agrupados es análogo a la desviación estándar poblacional σ y se la puede calcular con la Ecuación (26).

$$\sigma = \sqrt{\frac{\sum_{i=1}^{q} f_i \cdot (x_i - \mu)^2}{N}} = \sqrt{\sum_{i=1}^{q} fr_i \cdot (x_i - \mu)^2}\,, \qquad (26)$$

Código:

```
# Cálculo de la desviación estándar poblacional
des = ((tf['fr']*(tf['x'] - (tf['f']*tf['x']).sum())**2).sum())**(1/2)
print('%.4f' % des)
```

Salida:

16.3507

Nótese que, para los valores agrupados, la diferencia entre las desviaciones estándar muestral y poblacional (Ecuaciones (25) y (26)) es pequeña ($16{,}3671 - 16{,}3507 = 0{,}0164$), esto se debe a que el tamaño de la muestra de los caudales utilizados en estos ejemplos es de 500, que es mayor a 30; por lo que se cumple lo indicado con la Figura 4.1.

Cabe recordar que las mediciones de caudales pertenecen a una población infinita; por lo tanto, estos 500 datos no son exactamente la población; sin embargo, se les puede considerar como tales debido a que la desviación estándar, tanto de los valores independientes como de los valores agrupados calculados con las ecuaciones para la población y para la muestra arrojan valores similares.

Dicho esto, para el cálculo de la desviación estándar de los valores independientes de una muestra grande ($n > 30$), se recomienda utilizar siempre las ecuaciones (24) y (25), correspondientes al cálculo de la desviación estándar muestral.

4.3 Varianza

Es otra de las medidas de dispersión muy utilizadas, igual al cuadrado de la desviación estándar, por lo que a la varianza poblacional se le representa como σ^2, y como s^2 en el caso de la varianza muestral.

Para calcular la varianza poblacional de los valores independientes σ^2, se emplea la siguiente Ecuación:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}. \qquad (27)$$

Código:

Opción 1:

```
# Tamaño de la población
N = len(caudal)
# Cálculo de la varianza poblacional
var = ((caudal - caudal.mean())**2).sum()/N
print('%.4f' % var)
```

Salida:

258.0595

Opción 2:

```
# Cálculo de la varianza poblacional
var = caudal.var(ddof=0)
print('%.4f' % var)
```

Salida:

258.0595

Si se desarrolla el binomio al cuadrado de la Ecuación (27), se obtiene $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} {x_i}^2 - \mu^2 = \overline{x^2} - \bar{x}^2$, de la que cambiando la nomenclatura, la varianza $VAR(x)$, se la puede expresar a través de la media $E(x)$ de la siguiente manera:

$$VAR(x) = E(x^2) - E(x)^2 \,. \tag{28}$$

Código:

```
# Cálculo de la varianza poblacional
var = (caudal**2).mean() - (caudal.mean())**2
print('%.4f' % var)
```

Salida:

258.0595

Para calcular la varianza muestral de los valores independientes s^2, se emplea la siguiente Ecuación:

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}. \quad (29)$$

Código:

Opción 1:

```
# Tamaño de la muestra
n = len(caudal)
# Cálculo de la varianza muestral
var = ((caudal - caudal.mean())**2).sum()/(n-1)
print('%.4f' % var)
```

Salida:

258.5766

Opción 2:

```
# Cálculo de la varianza muestral
var = caudal.var()
print('%.4f' % var)
```

Salida:

258.5766

Para calcular de la varianza de los valores agrupados s de $n \leq 30$, se utiliza la Ecuación (30).:

$$s^2 = \frac{\sum_{i=1}^{q} f_i \cdot (x_i - \bar{x})^2}{n-1}. \quad (30)$$

Código:

```
# Tamaño de la muestra
n = len(caudal)
# Cálculo de la varianza muestral
var = (tf['f']*(tf['x'] - (tf['f']*tf['x']).sum())**2).sum()/(n-1)
print('%.4f' % var)
```

Salida:

267.8813

Para calcular de la varianza de los valores agrupados σ^2 de $n > 30$, se utiliza la siguiente Ecuación:

$$\sigma^2 = \sum_{i=1}^{q} fr_i \cdot (x_i - \mu)^2 \,. \tag{31}$$

Código:

```
# Cálculo de la varianza poblacional
var = (tf['fr']*(tf['x'] - (tf['f']*tf['x']).sum())**2).sum()
print('%.4f' % var)
```

Salida:

267.3455

Si se desarrolla el binomio al cuadrado de la Ecuación (31), se obtiene la Ecuación (32), misma que también se la puede expresar de acuerdo con la Ecuación (28).

$$\sigma^2 = \left(\sum_{i=1}^{q} fr_i \cdot {x_i}^2\right) - \left(\sum_{i=1}^{q} fr_i \cdot x_i\right)^2 \,. \tag{32}$$

Código:

```
# Cálculo de la varianza poblacional
var = (tf['fr']*tf['x']**2).sum() - ((tf['fr']*tf['x']).sum())**2
print('%.4f' % var)
```

Salida:

267.3455

4.4 Coeficiente de variación

Es una medida de la dispersión relativa de una muestra o de una población no negativa, que se calcula con las ecuaciones $CV = \frac{s}{\bar{x}} * 100\%$ o a su vez $CV = \frac{\sigma}{\mu} * 100\%$, respectivamente, cuyo resultado se recomienda que se redondee a un decimal (Triola, 2018, p. 106).

Código para el cálculo del coeficiente de variación muestral CV:

```
# Cálculo del coeficiente de variación muestral
cv = caudal.std(ddof=1)/caudal.mean()*100
print('%.1f' % cv,'%')
```

Salida:

4.3 %

Código para el cálculo del coeficiente de variación poblacional CV:

```
# Cálculo del coeficiente de variación poblacional
cv = caudal.std(ddof=0)/caudal.mean()*100
print('%.1f' % cv,'%')
```

Salida:

4.3 %

4.5 Cuantil

Es una medida de posición y corresponde al elemento de una muestra o población cuya frecuencia relativa acumulada tiene un valor específico. Por ejemplo, se puede encontrar de una muestra al elemento cuya frecuencia relativa acumulada sea 0,435 (43,5%), este valor a su vez tiene debajo de él a todos los elementos que en conjunto forman el 43,5% de toda la muestra (q43,5). Se le representa con una "q", del inglés *quantile* (Mendenhall et al., 2015).

Código:

Opción 1:

```
p = 43.5
# Ordenamiento de los datos
ordenados = sorted(caudal)
# Cálculo de la posición del cuantil
posicion = p/100 * (len(ordenados) - 1)
# Interpolación si es necesario
if p <= 100:
    if posicion.is_integer():
        q = ordenados[int(posicion)]
    else:
        entero = int(posicion)
        decimal = posicion - entero
                q = (1 - decimal) * ordenados[entero] + decimal *
ordenados[entero+1]
else:
    q = ordenados[len(ordenados) - 1]
print(f'q{p}:', q)
```

Salida:

q43.5: 367.8

Opción 2:

```
p = 43.5
# Importación de la librería numpy
import numpy as np
# Cálculo del cuantil NO mayor de 100
P = np.percentile(caudal, p)
print(f'q{p}:', P)
```

Salida:

q43.5: 367.8

El cuantil de los valores agrupados q, de un porcentaje cualquiera p, se calcula con la siguiente Ecuación:

$$q = L_{inf_i} + \frac{n * p - F_{i-1}}{f_i} \cdot a\,, \tag{33}$$

donde q es el cuantil buscado, p es el porcentaje donde está el cuantil que se desea encontrar, L_{inf_i} es el límite inferior de la tabla de frecuencias donde se encuentra el cuantil buscado, F_{i-1} es la frecuencia absoluta acumulada de la clase anterior de donde se encuentra el cuantil buscado, f_i es la frecuencia absoluta donde se encuentra el cuantil buscado y a es el ancho de clases.

Código:

```
p = 43.5
# Búsqueda del índice de la clase donde está el cuantil
k = min([i for i, valor in enumerate(tf['Fra']) if valor >= p/100])
# Cálculo del cuantil buscado
q = tf['Linf'][k]+(n*p/100-tf['Fa'][k-1])/tf['f'][k]*a
print(f'q{p}: %.1f' % q)
```

Salida:

q43.5: 366.4

Los cuantiles más comunes son los percentiles, deciles y los cuartiles, cuyo cálculo se realiza con los mismos métodos utilizados para los cuantiles.

4.5.1 Percentil

Los percentiles (P, del inglés *Percentile*) corresponden a los elementos de una muestra que la dividen en 100 partes iguales; por lo tanto, el valor que contiene al 1% del total se le conoce como el primer percentil (P1).

Código para encontrar los percentiles 45 y 85 (P45 y P85) de los valores independientes:

```
# Importación de la librería numpy
import numpy as np
# Cálculo de los percentiles P45 y P85
P45 = np.percentile(caudal, 45)
P85 = np.percentile(caudal, 85)
print(f'P45: %.1f' % P45)
print(f'P85: %.1f' % P85)
```

Salida:

P45: 368.2

P85: 386.7

Código para encontrar los percentiles 45 y 65 (P45 y P65) de los valores agrupados:

```
p1 = 45
p2 = 85
# Búsqueda y cálculo de P45
k1 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p1/100])
P45 = tf['Linf'][k1]+(n*p1/100-tf['Fa'][k1-1])/tf['f'][k1]*a
# Búsqueda y cálculo de P45
k2 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p2/100])
P85 = tf['Linf'][k2]+(n*p2/100-tf['Fa'][k2-1])/tf['f'][k2]*a
print(f'P45: %.1f' % P45)
print(f'P85: %.1f' % P85)
```

Salida:

P45: 367.1

P85: 387.4

4.5.2 Decil

Los deciles (D, del inglés *Decile*) corresponden a los elementos de una muestra que la dividen en 10 partes iguales; por lo tanto, el valor que contiene al 10% del total (P10) se le conoce como el primer decil (D1).

Código para encontrar los deciles 4 y 8 (D4 y D8) de los valores independientes:

```
# Importación de la librería numpy
import numpy as np
# Cálculo de los deciles D4 y D8
D4 = np.percentile(caudal, 0.40)
D8 = np.percentile(caudal, 0.80)
print(f'D4: %.1f' % D4)
print(f'D8: %.1f' % D8)
```

Salida:

D4: 365.8

D8: 383.7

Código para encontrar los deciles 4 y 8 (D4 y D8) de los valores agrupados:

```
p1 = 40
p2 = 80
# Búsqueda y cálculo de P45
k1 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p1/100])
D4 = tf['Linf'][k1]+(n*p1/100-tf['Fa'][k1-1])/tf['f'][k1]*a
# Búsqueda y cálculo de P45
k2 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p2/100])
D8 = tf['Linf'][k2]+(n*p2/100-tf['Fa'][k2-1])/tf['f'][k2]*a
print(f'D4: %.1f' % D4)
print(f'D8: %.1f' % D8)
```

Salida:

D4: 365.0

D8: 383.7

4.5.3 Cuartil

Los cuartiles (Q, del inglés *Quartile*) corresponden a los elementos de una muestra que la dividen en 4 partes iguales; por lo tanto, el valor que contiene al 25% del total (P25) se le conoce como el primer cuartil (Q1).

Código para encontrar los cuartiles 1, 2 y 3 (Q1, Q2 y Q3) de los valores independientes:

```
# Importación de la librería numpy
import numpy as np
# Cálculo de los cuartiles Q1, Q2 y Q3
Q1 = np.percentile(caudal, 25)
Q2 = np.percentile(caudal, 50)
Q3 = np.percentile(caudal, 75)
print(f'Q1: %.1f' % Q1)
print(f'Q2: %.1f' % Q2)
print(f'Q3: %.1f' % Q3)
```

Salida:

Q1: 358.7

Q2: 369.9

Q3: 380.6

Código para encontrar los cuartiles 1, 2 y 3 (Q1, Q2 y Q3) de los valores agrupados:

```
p1 = 25
p2 = 50
p3 = 75
# Búsqueda y cálculo de P45
k1 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p1/100])
Q1 = tf['Linf'][k1]+(n*p1/100-tf['Fa'][k1-1])/tf['f'][k1]*a
```

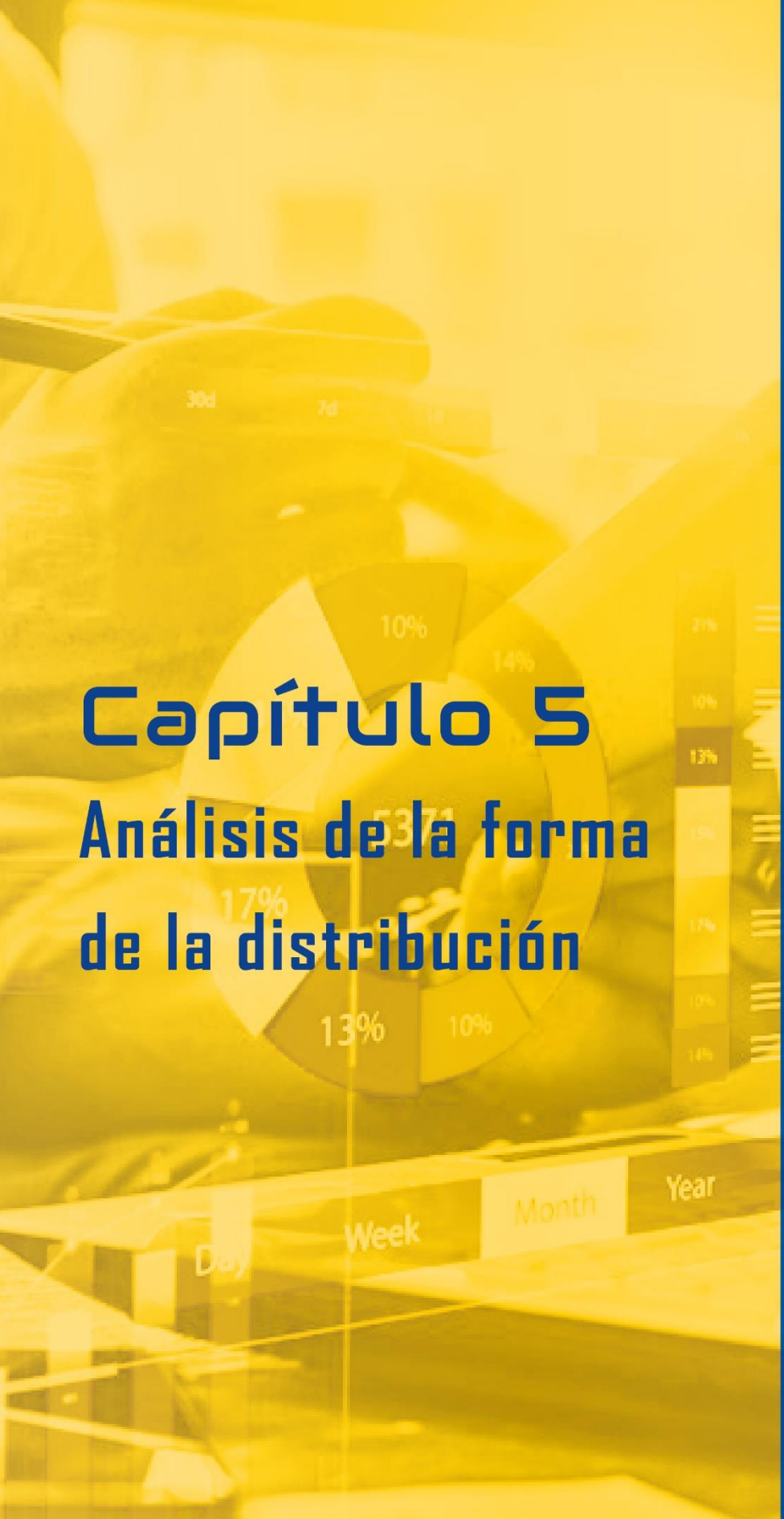
```
# Búsqueda y cálculo de P45
k2 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p2/100])
Q2 = tf['Linf'][k2]+(n*p2/100-tf['Fa'][k2-1])/tf['f'][k2]*a
# Búsqueda y cálculo de P45
k3 = min([i for i, valor in enumerate(tf['Fra']) if valor >= p3/100])
Q3 = tf['Linf'][k3]+(n*p3/100-tf['Fa'][k3-1])/tf['f'][k3]*a
print(f'Q1: %.1f' % Q1)
print(f'Q2: %.1f' % Q2)
print(f'Q3: %.1f' % Q3)
```

Salida:

Q1: 357.7

Q2: 369.2

Q3: 381.1



Capítulo 5

Análisis de la forma de la distribución

Capítulo

5

5. Análisis de la forma de la distribución

5.1 Normalidad de los datos

Los datos se consideran normales, cuando se ajustan a la distribución normal, que ocurre cuando los valores de la frecuencia relativa se aproximan a la función (Ecuación (34)) que forma la llamada campana de Gauss (llamada así porque Gauss formuló su Ecuación, aunque Moivre la descubrió) (Pértegas & Pita, 2001).

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}. \qquad (34)$$

5.1.1 Método analítico

El método más difundido para evaluar la aproximación de los datos a una distribución normal es la prueba de Shapiro. Que se lleva a cabo mediante la librería *scipy.stats*.

Código:

```
# Importación de la librería scipy.stats
import scipy.stats as stats
datos = [10, 20, 20, 30, 30, 30, 40, 40, 50, 60, 70, 80]
# Shapiro-Wilk test
estadistico, p_valor = stats.shapiro(datos)
print("Estadístico: %.5f, p valor: %.5f" % (estadistico, p_valor))
if p_valor < 0.05:
   print("Los datos no siguen una distribución normal (asimétricos).")
else:
   print("Los datos siguen una distribución normal (simétricos).")
```

Salida:

Estadístico: 0.94492, p valor: 0.56436

Los datos siguen una distribución normal (simétricos).

A pesar de que los datos del ejemplo se distribuyen normalmente, no significa que existe una coincidencia exacta, esto se puede apreciar comparando la media con la mediana, mismos que tiene los valores de 40 y 35 respectivamente, donde se observa que $\bar{x} > Me$; por lo tanto, la distribución tiene sesgo hacia la derecha.

Código:

```
# Importación de la librería pandas
import pandas as pd
datos = [10, 20, 20, 30, 30, 30, 40, 40, 50, 60, 70, 80]
# Conversión de la lista datos al formato Series
datos = pd.Series(datos)
print("Media: %.5f, Me: %.5f" % (datos.mean(), datos.median()))
```

Salida:

Media: 40.00000, Me: 35.00000

5.1.2 Método gráfico

Gráficamente también se puede observar esta falta de coincidencia comparando el histograma de frecuencias relativas de los datos con la campana de Gauss (Figura 5.1) o mediante el llamado gráfico Q-Q (Figura 5.2) en el que algunos puntos de dispersión no coinciden con la recta.

Código del histograma de frecuencias relativas y la campana de Gauss:

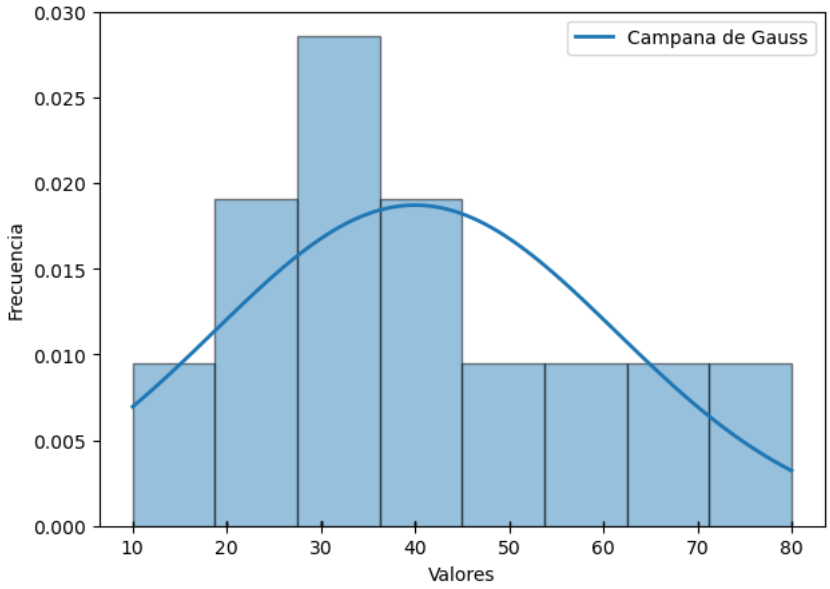
```
# Importación de las librerías
import pandas as pd
import matplotlib.pyplot as plt

datos = [10, 20, 20, 30, 30, 30, 40, 40, 50, 60, 70, 80]

# Conversión de la lista datos al formato Series
datos2 = pd.Series(datos)
# Valores en "x" para el gráfico de densidad teórica
x = np.linspace(min(datos), max(datos), num=100)
# Valores en "y" para el gráfico de densidad teórica (campana de Gauss)
y = stats.norm.pdf(x, datos2.mean(), datos2.std(ddof=0))
# Histograma de frecuencias
fig, ax = plt.subplots(figsize=(7,4))
```

```
ax.plot(x, y, linewidth=2, label='Campana de Gauss')
ax.hist(datos, density=True, bins=8, color="#3182bd", alpha=0.5,
ec="black")
ax.plot(datos, np.full_like(datos2, -0.002), '|k', markeredgewidth=1)
ax.set_xlabel('Valores')
ax.set_ylabel('Frecuencia')
ax.legend();
```

Salida:

Figura 5.1

Comparación del histograma de frecuencias relativas con la campana de Gauss.



Nota. Los autores**.**

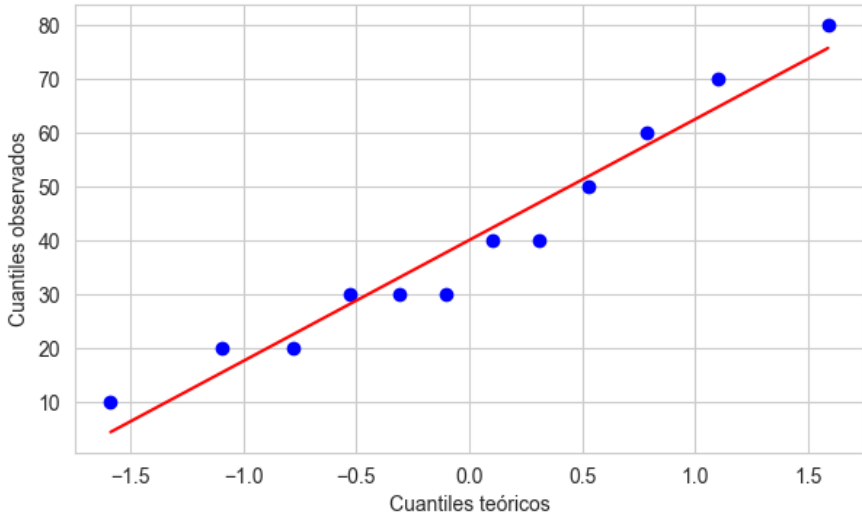
Código del diagrama Q-Q:

```
# Importación de las librerías
import matplotlib.pyplot as plt
import statsmodels.api as sm

datos = [10, 20, 20, 30, 30, 30, 40, 40, 50, 60, 70, 80]
datos = pd.Series(datos)

# Crea el gráfico Q-Q
fig, ax = plt.subplots(figsize=(7,4))
stats. probplot(datos, dist='norm', plot=plt)
ax.set_xlabel('Cuantiles teóricos')
ax.set_ylabel('Cuantiles observados')
ax.set_title('')
```

Salida:

Figura 5.2

Diagrama Q-Q de la muestra datos.



Nota. Los autores**.**

5.2 Simetría y asimetría

Un valor de asimetría (*skewness*) de 0 (cero) indica una distribución perfectamente simétrica $\bar{x} = Me$ (Figura 3.1). Un valor de asimetría positiva indica que la cola de la distribución es más larga en el lado derecho (distribución sesgada a la derecha $\bar{x} > Me$, como se indica en la Figura 3.2), mientras que un valor de asimetría negativa indica que la cola de la distribución es más larga en el lado izquierdo (distribución sesgada a la izquierda $\bar{x} < Me$, como se indica en la Figura 3.3). Cuanto mayor es el valor absoluto de la asimetría, más asimétrica es la distribución (Amat, 2021). Para calcular la curtosis de un conjunto de datos en formato *DataFrame* o Series, se puede usar la función *skew*().

Código:

```
# Importación de la librería pandas
import pandas as pd

datos = [10, 20, 20, 30, 30, 30, 40, 40, 50, 60, 70, 80]
datos = pd.Series(datos)

# Cálculo del coeficiente de simetría
simetria = datos.skew()
print("Coeficiente de simetría: %.5f" % simetria)
```

Salida:

Coeficiente de simetría: 0.60788

Como el valor del coeficiente de asimetría es positivo y menor a 1 (0,60788), se ratifica que existe un ligero sesgo hacia la derecha.

5.3 Curtosis

La curtosis es una medida del "pico" de una distribución. Una distribución normal tiene una curtosis de 3, mientras que una distribución con un pico más alto que una distribución normal tiene una curtosis positiva, y una distribución con un pico más bajo tiene una curtosis negativa. Una distribución con una curtosis mayor que 3 se llama *leptokurtic* y sus datos son más dispersos que la distribución normal, mientras que una distribución con una curtosis menor que 3 se llama *platykurtic* y sus datos son menos dispersos que la distribución normal. Para calcular la curtosis de un conjunto de datos en formato *DataFrame* o Series, se puede usar la función *kurtosis*() (Statologos, 2021).

Código:

```
# Importación de la librería pandas
import pandas as pd

datos = [10, 20, 20, 30, 30, 30, 40, 40, 50, 60, 70, 80]
datos = pd.Series(datos)

# Cálculo de la curtosis
curtosis = datos.kurtosis()
print("Curtosis: %.5f" % curtosis)
```

Salida:

Curtosis: -0.44880

Como la curtosis de la muestra es menor que 3 ($-0{,}44880$), se puede concluir que la distribución es platicúrtica, o lo que es lo mismo, la dispersión de los datos es menor que la distribución normal.

6

Capítulo 6

Manejo de datos faltantes y atípicos

Capítulo 6

6. Manejo de datos faltantes y atípicos

6.1 Imputación de datos faltantes

Existen diversas técnicas para la imputación de datos faltantes dentro de las cuales las más comunes son eliminación de los registros con datos faltantes, imputación simple y k-vecinos (Rodríguez, 2021).

6.1.1 Eliminación de los registros con datos faltantes

La técnica de eliminación de los registros con datos faltantes consiste en eliminar los registros que contienen datos faltantes. Sin embargo, puede no ser adecuada si los datos de las otras columnas son importantes o si la cantidad de datos faltantes es grande.

Código:

```
# Creación de un DataFrame con datos faltantes
datos = {'A': [1, 2, None, 4],
        'B': [None, 2, 3, 4],
        'C': [1, 2, 3, 4]}

datos = pd.DataFrame(datos)
print('Datos originales:\n', datos, '\n')

# Eliminación de filas con valores faltantes
sin_nulos = datos.dropna()
print('Datos sin filas con datos faltantes:\n', sin_nulos, '\n')

# Eliminación de columnas con valores faltantes
sin_nulos_columnas = datos.dropna(axis=1)
print('Datos sin columnas con datos faltantes:\n', sin_nulos_columnas)
```

Salida:

Datos originales:

	A	B	C
0	1.0	NaN	1
1	2.0	2.0	2
2	NaN	3.0	3
3	4.0	4.0	4

Datos sin filas con datos faltantes:

	A	B	C
1	2.0	2.0	2
3	4.0	4.0	4

Datos sin columnas con datos faltantes:

	C
0	1
1	2
2	3
3	4

6.1.2 Imputación simple

La técnica de imputación simple consiste en reemplazar los valores faltantes por la media, la mediana o la moda de la variable de la columna correspondiente o por alguna constante que resulte conveniente para los datos (Valverde et al., 2023, p. 209).

Código:

```
# Importación de la librería pandas
import pandas as pd

# Creación de un DataFrame con datos faltantes
datos = {'A': [1, 5, 6, None],
        'B': [1, 5, 6, None],
        'C': [1, 5, 6, None]}
datos = pd.DataFrame(datos)
datos2 = datos.copy()
print('Datos originales:\n', datos, '\n')

# Rellenar valores faltantes con un valor constante, por ejemplo, 0
relleno_cte = datos.fillna(0)
print('Relleno con el 0:\n', relleno_cte, '\n')
```

```
# Rellenar valores faltantes en columna 'A' con la media de esa columna
media_A = datos['A'].mean()
datos['A'].fillna(media_A, inplace=True)
print('Relleno de la columna A con su media:\n', datos, '\n')

# Rellenar valores faltantes en columna 'B' con la mediana de esa columna
mediana_B = datos['B'].median()
datos['B'].fillna(mediana_B, inplace=True)
print('Relleno de la columna B con su mediana:\n', datos, '\n')

# Rellenar valores faltantes en columna 'C' con la moda de esa columna
moda_C = datos['C'].mode()[0]
datos['C'].fillna(moda_C, inplace=True)
print('Relleno de la columna C con su moda:\n', datos, '\n')

# Rellenar valores faltantes con la mediana de esa columna
mediana = datos2.median()
datos2.fillna(mediana, inplace=True)
print('Relleno de todas las columnas con sus respectivas medianas:\n',
datos2, '\n')
```

Salida:

Datos originales:

	A	B	C
0	1.0	1.0	1.0
1	5.0	5.0	5.0
2	6.0	6.0	6.0
3	NaN	NaN	NaN

Relleno con el 0:

	A	B	C
0	1.0	1.0	1.0

```
1   5.0   5.0   5.0
2   6.0   6.0   6.0
3   0.0   0.0   0.0
```

Relleno de la columna A con su media:

```
     A    B     C
0   1.0   1.0   1.0
1   5.0   5.0   5.0
2   6.0   6.0   6.0
3   4.0   NaN   NaN
```

Relleno de la columna B con su mediana:

```
     A    B    C
0   1.0   1.0   1.0
1   5.0   5.0   5.0
2   6.0   6.0   6.0
3   4.0   5.0   NaN
```

Relleno de la columna C con su moda:

```
     A     B    C
0   1.0   1.0   1.0
1   5.0   5.0   5.0
2   6.0   6.0   6.0
3   4.0   5.0   1.0
```

Relleno de todas las columnas con sus respectivas medianas:

```
     A    B    C
0   1.0   1.0   1.0
1   5.0   5.0   5.0
2   6.0   6.0   6.0
3   5.0   5.0   5.0
```

6.1.3 k-vecinos

La técnica k-vecinos consiste en reemplazar los valores nulos con la media de sus vecinos, lo que posiblemente puede producir un valor más parecido al real.

Código:

```
# Importación de las librerías
import pandas as pd
from sklearn.impute import KNNImputer

datos = {'A': [1, 2, None, 4, 5],
        'B': [None, 2, 3, 4, None],
        'C': [1, None, None, 4, 5]}
datos = pd.DataFrame(datos)
print('Datos originales:\n', datos, '\n')

# n_neighbors es el número de vecinos requeridos
imputer = KNNImputer(n_neighbors = 2)
datos_imputados = imputer.fit_transform(datos)
datos_imputados = pd.DataFrame(datos_imputados,
columns=datos.columns)
print('Relleno con la media de 2 vecinos:\n', datos_imputados)
```

Salida:

```
Datos originales:
     A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  NaN
```

```
2   NaN   3.0     NaN
3   4.0    4.0    4.0
4   5.0    NaN    5.0
```

Relleno con la media de 2 vecinos:

```
     A    B    C
0   1.0  3.0  1.0
1   2.0  2.0  2.5
2   3.0  3.0  2.5
3   4.0  4.0  4.0
4   5.0  3.0  5.0
```

6.2 Identificación de valores atípicos

Las técnicas más comunes para identificar valores atípicos (*outliers*) son el método del rango intercuartílico y el método Z-Score.

6.2.1 Método del rango intercuartílico

El método del rango intercuartílico (RIC), se basa en establecer un límite inferior y otro superior iguales a $Q1 - 1{,}5 \cdot RIC$ y $Q3 + 1{,}5 \cdot RIC$, respectivamente, donde $Q1$ es el primer cuartil, $Q3$ es el tercer cuartil y RIC es el rango intercuartílico igual a $Q3 - Q1$. Todos los valores que estén por debajo o por encima de estos límites, son los valores atípicos (Valverde et al., 2023, p. 212).

Código:

```
# Importación de la librería numpy
import numpy as np

# Datos de ejemplo
datos = [90, 100, 70, 80, 180, 60, 110, 80, 70, 80, 90, 90, 100, 200]

# Determinación de los límites
Q1 = np.percentile(datos, 25)
Q3 = np.percentile(datos, 75)
RIC = Q3 - Q1
L_inf = Q1 - 1.5 * RIC
L_sup = Q3 + 1.5 * RIC
# Identificación de los valores atípicos
atipicos = [x for x in datos if x < L_inf or x > L_sup]

print("Valores atípicos:", atipicos)
```

Salida:

```
Valores atípicos: [180, 200]
```

6.2.2 Método Z-Score

En el método Z-Score, los límites miden cuántas desviaciones estándar un punto de datos está lejos de la media. Los puntos de datos por arriba o por debajo de los límites se consideran atípicos. Puedes calcular el Z-Score asociado a una

probabilidad; donde por ejemplo, el umbral de 1,64 genera un rango que contiene a los valores de la población con el 90% de probabilidad; por lo tanto, aproximadamente el 10% de los datos (5% por arriba y 5% por debajo) se consideran atípicos.

Código:

```
# Importación de la librería scipy.stats
import scipy.stats as stats

# Datos de ejemplo
datos = [90, 100, 70, 80, 180, 60, 110, 80, 70, 80, 90, 90, 100, 200]

# Estandarización de los valores de la muestra datos
z_scores = np.abs(stats.zscore(datos))
# umbral para identificar valores atípicos
umbral = 1.64
# Identificación de los valores atípicos
valores_atipicos = [dato for dato, z in zip(datos, z_scores) if z>umbral]

print("Valores atípicos:", valores_atipicos)
```

Salida:

Valores atípicos: [180, 200]

6.3 Tratamiento de valores atípicos

El tratamiento de valores atípicos debe basarse en un entendimiento profundo del dominio y en el contexto del problema. No todos los valores atípicos deben ser eliminados; algunos pueden ser representativos de eventos raros o significativos en los datos. En el caso de que el respectivo análisis determine que se debe realizar el tratamiento de los valores atípicos, se pueden emplear el método de eliminación o el de imputación.

6.3.1 Método de eliminación

El método de eliminación consiste en eliminar los valores atípicos que se han identificado claramente como errores o datos irrelevantes.

Código:

```
# Importación de la librería
import numpy as np
import pandas as pd

# Datos de ejemplo con un valor atípico
datos = [90, 100, 70, 80, 180, 60, 110, 80, 70, 80, 90, 90, 100, 200]
datos = pd.Series(datos)
```

```
# Mostrar la descripción de los datos originales
print("Datos originales:\n",datos.describe())

# Identificación de los valores atípicos
Q1 = np.percentile(datos, 25)
Q3 = np.percentile(datos, 75)
ric = Q3 - Q1
umbral_inferior = Q1 - 1.5 * ric
umbral_superior = Q3 + 1.5 * ric
valores_atipicos = [x for x in datos if x < umbral_inferior or x > umbral_superior]
print("\nValores atípicos:", valores_atipicos, '\n')

# Eliminación de los valores atípicos
datos_sin_atipico = datos
for i in range(len(valores_atipicos)):
    datos_sin_atipico = [x for x in datos_sin_atipico if x != valores_atipicos[i]]
datos_sin_atipico=pd.Series(datos_sin_atipico)

# Mostrar la descripción de los datos sin atípicos
print("datos sin atipicos:\n",datos_sin_atipico.describe())

# Shapiro-Wilk test
estadistico, p_valor = stats.shapiro(datos_sin_atipico)
print("Estadístico=%.5f, p valor=%.5f" % (estadistico, p_valor))
```

Datos originales:

```
count     14.000000
mean     100.000000
std       40.572822
min       60.000000
25%       80.000000
50%       90.000000
75%      100.000000
```

```
max       200.000000
dtype:    float64

Valores atípicos: [180, 200]

datos sin atípicos:
count      12.000000
mean       85.000000
std        14.459976
min        60.000000
25%        77.500000
50%        85.000000
75%        92.500000
max       110.000000
dtype:    float64
Estadístico=0.96869, p valor=0.89666
```

6.3.2 Método de imputación

Con el método de imputación se sustituyen a aquellos valores atípicos que por algún motivo no se pueden eliminar pero que se sospecha que son errores, por la mediana del resto de los datos.

Código:

```
import numpy as np
import pandas as pd

# Datos de ejemplo con un valor atípico
```

```
datos = [90, 100, 70, 80, 180, 60, 110, 80, 70, 80, 90, 90, 100, 200]
datos = pd.Series(datos)

# Mostrar la descripción de los datos originales
print("Datos originales:\n", datos.describe())

# Identificación de los valores atípicos
Q1 = np.percentile(datos, 25)
Q3 = np.percentile(datos, 75)
ric = Q3 - Q1
umbral_inferior = Q1 - 1.5 * ric
umbral_superior = Q3 + 1.5 * ric
valores_atipicos = [x for x in datos if x < umbral_inferior or x > umbral_superior]
print("\nValores atípicos:", valores_atipicos, '\n')

# Calcular la mediana de los datos sin el valor atípico
datos_sin_atipico = datos
for i in range(len(valores_atipicos)):
   datos_sin_atipico = [x for x in datos_sin_atipico if x != valores_atipicos[i]]
mediana_sin_atipico = np.median(datos_sin_atipico)

# Imputar el valor atípico con la mediana
datos_imputados = datos
for i in range(len(valores_atipicos)):
    datos_imputados = [mediana_sin_atipico if x == valores_atipicos[i] else x for x in datos_imputados]
datos_imputados=pd.Series(datos_imputados)

# Mostrar la información general
print("Datos imputados:\n",datos_imputados.describe())

# Shapiro-Wilk test
estadistico, p_valor = stats.shapiro(datos_imputados)
print("Estadístico=%.5f, p valor=%.5f" % (estadistico, p_valor))
```

Salida:

```
Datos originales:
count       14.000000
mean     100.000000
std         40.572822
min         60.000000
25%         80.000000
50%         90.000000
75%       100.000000
max       200.000000
dtype:    float64

Valores atípicos: [180, 200]

Datos imputados:
count       14.000000
mean        85.000000
std         13.301243
min         60.000000
25%         80.000000
50%         85.000000
75%         90.000000
max       110.000000
dtype:    float64
Estadístico=0.97484, p valor=0.93380
```

Nótese que, con el método de imputación, se obtiene una muestra con un p valor mayor que con el método de eliminación. Esto no ocurre en todos los casos por lo que se recomienda analizar los efectos del tratamiento de los valores atípicos y seleccionar el método que más convenga, que podría ser el que aproxime a la muestra de mejor manera a una distribución normal.

7

Capítulo 7

Elaboración de gráficos estadísticos

Capítulo 7

7. Elaboración de gráficos estadísticos

En función del tipo de gráfico se requieren distintas librerías y clases de datos; por lo tanto, estas se definen al inicio de cada código, a excepción de los histogramas y diagramas de cajas en los que se emplea los datos importados a *Python Jupyter* que se indican en el Capítulo 2.

7.1 Histograma de frecuencias y gráfico de densidad teórica

El histograma de frecuencias es una herramienta importante porque permite visualizar la distribución de un conjunto de datos de manera gráfica y resumida, lo que facilita la identificación de patrones, características inusuales y tendencias. Además, permite identificar la forma de la distribución, lo que puede ser útil para seleccionar el modelo estadístico adecuado para el análisis. Si se cuenta con dos o más muestras, se puede comparar sus distribuciones para identificar diferencias o similitudes entre ellas (Johnson, 2012, pp. 17-20).

7.1.1 Histograma de frecuencias relativas

Tradicionalmente, para elaborar el histograma de frecuencias relativas, se parte de la tabla de frecuencias; sin embargo, en la actualidad se prefiere utilizar la librería *matplotlib* para obtener el histograma directamente de la muestra de valores independientes, como se indica a continuación:

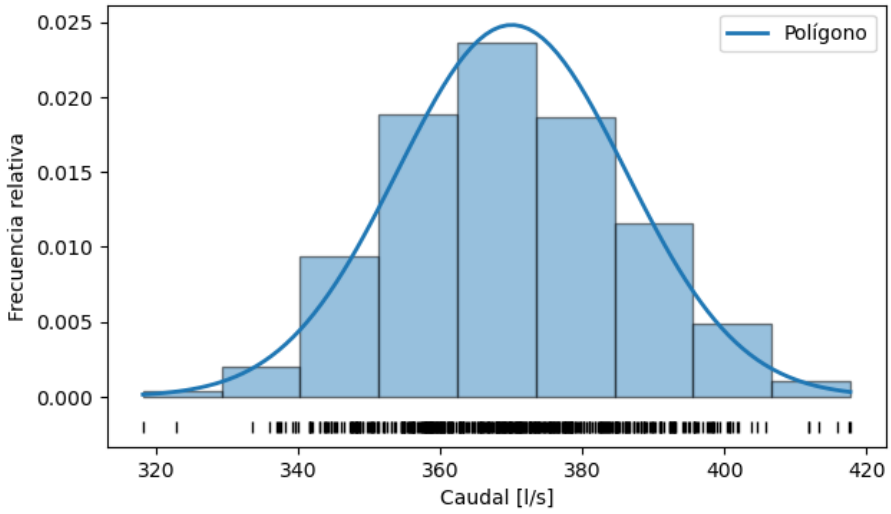
Código:

```
# Importación de las librerías
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Conversión del DataFrame caudal al arreglo arr
arr = np.array(caudal)
# Valores en "x" para el gráfico de densidad teórica
x_2 = np.linspace(caudal.min(), caudal.max(), num=100)
# Valores en "y" para el gráfico de densidad teórica (campana de Gauss)
y_2 = stats.norm.pdf(x_2, caudal.mean(), caudal.std(ddof=0))

# Histograma de frecuencias y gráfico de densidad teórica
fig, ax = plt.subplots(figsize=(7,4))
ax.plot(x_2, y_2, linewidth=2, label='Polígono')
ax.hist(caudal, density=True, bins=9, color="#3182bd", alpha=0.5,
ec="black")
ax.plot(arr, np.full_like(caudal, -0.002), '|k', markeredgewidth=1)
ax.set_xlabel('Caudal [l/s]')
ax.set_ylabel('Frecuencia relativa')
ax.legend();
```

Salida:

Figura 7.1

Histograma de frecuencias relativas de la muestra de caudales con $q = 9$.



Nota. Los autores**.**

Nótese que la Figura 7.1 no tiene un título en la parte superior, debido a que para ser usado en un documento formal, no se recomienda; sin embargo, en el caso de requerirlo, se debe intercalar entre la última y la penúltima línea del código anterior, la siguiente:

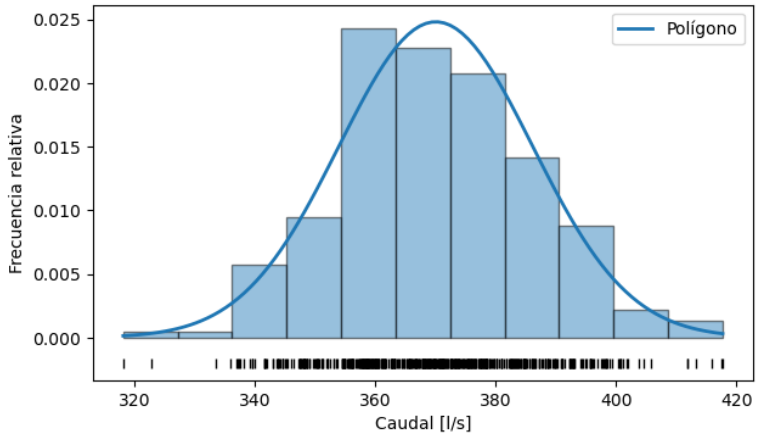
```
ax.set_title('Histograma de frecuencias')
```

En este caso, el histograma generado tiene 9 clases (el número de clases se estableció para la elaboración de la tabla de frecuencias y corresponde al número de barras del histograma); a pesar de que el cálculo con la ley de Sturges da como resultado 10 clases ($q = 1 + log_2 500 \approx 10$).

Como se recomienda de que el número de clases sea impar, las alternativas son 9 y 11. Para seleccionar el valor más adecuado, se prueba con los posibles candidatos modificando el parámetro *bins* de la función *hist*(); de ahí se observa que si *bins = 11*, el histograma se muestra algo deforme, como se indica en la siguiente Figura.

Figura 7.2

Histograma de frecuencias relativas de la muestra de caudales con $q = 11$.



Nota. Los autores**.**

7.1.2 Histograma de frecuencias acumuladas

Por otro lado, para graficar el histograma de frecuencias acumuladas, a la función *hist()*, se le debe insertar el parámetro *cumulative = True*, y en lugar de la lista *y_2*, se crea la lista *y_3* con los valores de probabilidades acumulada mediante la función *stats.norm.cdf* para posteriormente graficar la función de probabilidades acumulada conocida como Ojiva mediante la función *plot*, como se indica a continuación:

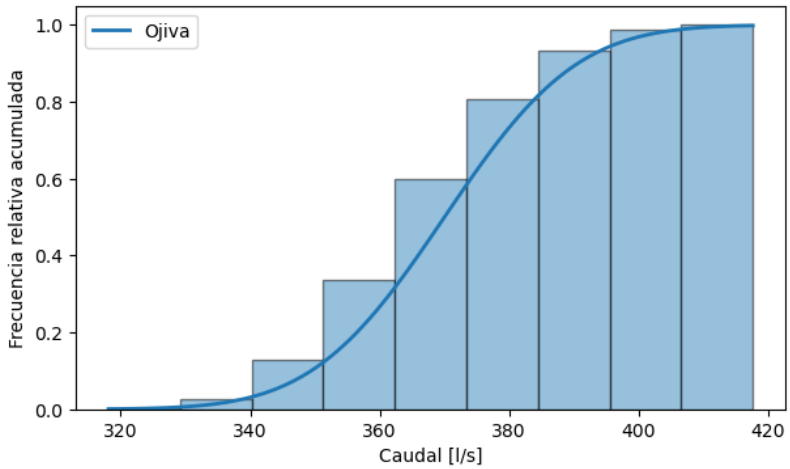
Código:

```
# Importación de las librerías
import numpy as np
import matplotlib.pyplot as plt

# Conversión del DataFrame caudal al arreglo arr
arr = np.array(caudal)
# Valores en "x" para el gráfico de densidad teórica
x_2 = np.linspace(caudal.min(), caudal.max(), num=100)
# Valores en "y" para el gráfico de probabilidades acumulada teórica
y_3 = stats.norm.cdf(x_2, caudal.mean(), caudal.std(ddof=0))

# Histograma de frecuencias y gráfico de densidad teórica
fig, ax = plt.subplots(figsize=(7,4))
ax.plot(x_2, y_3, linewidth=1.5, color="black", label='Ojiva')
ax.hist(caudal, density=True, bins=q, color="#3182bd", alpha=0.5,
ec="black", cumulative=True)
ax.set_xlabel('Caudal [l/s]')
ax.set_ylabel('Frecuencia relativa acumulada')
ax.legend();
```

Salida:

Figura 7.3

Histograma de frecuencia relativa acumulada de la muestra de caudales con $q = 9$.



Nota. Los autores**.**

7.2 Diagrama de caja

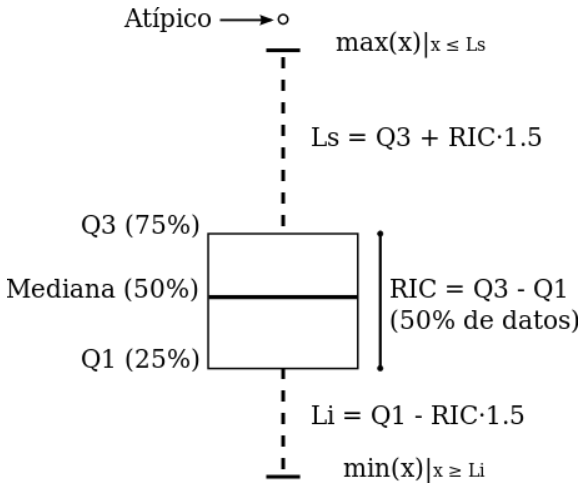
El diagrama de cajas o en inglés *box plot*, es una herramienta gráfica utilizada para identificar la centralización y dispersión de los datos, detectar valores atípicos que pueden afectar el análisis y comparar distribuciones entre diferentes grupos de datos. Como de observa en la Figura 7.4, este diagrama proporciona información sobre la mediana, los cuartiles y los posibles valores atípicos (*outliers*) presentes en los datos (Mendenhall et al., 2015, p. 77; Triola, 2018, p. 119).

Sus elementos son:

1. **Caja (*Box*):** es la caja en el centro del diagrama que representa el rango intercuartílico (RIC), que es la distancia entre el primer cuartil (Q1) y el tercer cuartil (Q3). La mediana (Q2) se muestra como una línea dentro de la caja.
2. **Bigotes (*Whiskers*):** son las dos líneas que se extienden desde la caja, hasta 1,5 veces el RIC.
3. **Valores Atípicos (*Outliers*):** son los puntos individuales fuera del rango de los bigotes pueden ser indicativos de datos inusuales o errores de medición.

Figura 7.4

Diagrama de cajas.



Nota. Adaptado de Nube de datos (2015).

El Código mediante la librería *matplotlib.pyplot* para elaborar el diagrama de cajas de la muestra caudal, se puede escribir de la siguiente manera:

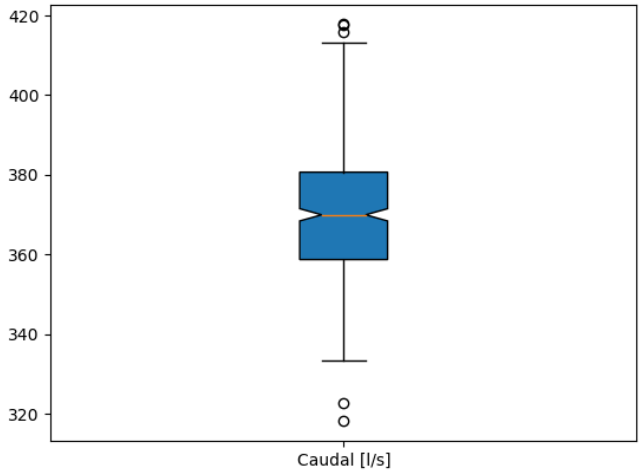
Código:

```
# Importación de la librería matplotlib.pyplot
import matplotlib.pyplot as plt

# Elaboración del diagrama de cajas
plt.boxplot(caudal, vert=True, patch_artist=True, notch ='True',
labels=["Caudal [l/s]"]);
```

Salida:

Figura 7.5

Diagrama de cajas de la muestra de caudal



Nota. Los autores**.**

Si se desea comparar algunas muestras mediante un diagrama de cajas, se debe agrupar los datos en una sola lista o en una tabla de datos por medio de un *DataFrame* de pandas, como se indica a continuación.

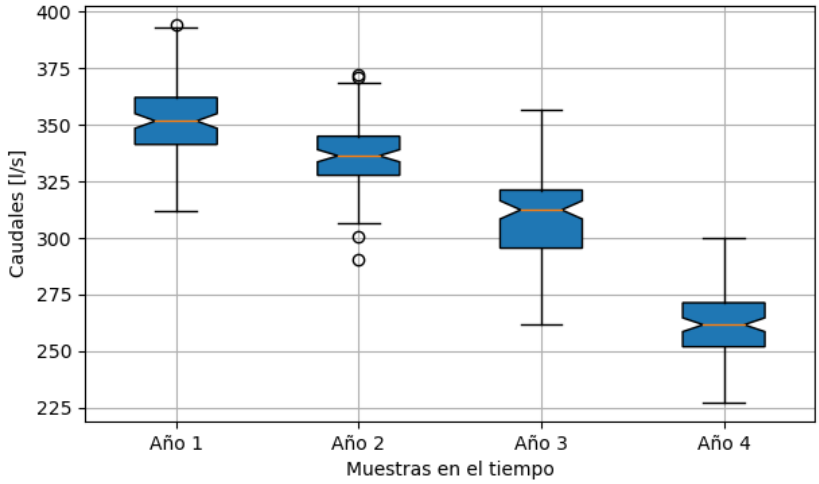
Código:

```
# Importación de las librerías
import matplotlib.pyplot as plt
import numpy as np

# Creación de 4 muestras aleatorias con la semilla 10
np.random.seed(10) # semilla = 10
muestra1 = np.random.normal(350, 18, 100)
muestra2 = np.random.normal(335, 15, 100)
muestra3 = np.random.normal(310, 20, 100)
muestra4 = np.random.normal(260, 15, 100)

# Agrupación de las muestras en una lista
muestras = [muestra1, muestra2, muestra3, muestra4]

# Elaboración del diagrama de cajas para comparar las muestras
plt.figure(figsize=(7, 4))
plt.boxplot(muestras, labels=['Año 1', 'Año 2', 'Año 3', 'Año 4'],
        vert=True, patch_artist=True, notch ='True')
plt.xlabel('Muestras en el tiempo')
plt.ylabel('Caudales [l/s]')
plt.grid(True)
plt.show()
```

Salida:

Figura 7.6

Comparación de 4 muestras mediante un diagrama de cajas.



Nota. Los autores**.**

El diagrama de cajas indica que el caudal disminuye en función del paso del tiempo y aunque los rangos comprendidos entre el máximo y mínimo de los caudales de los distintos años se traslapan, las medianas disminuyen progresivamente.

Para conocer con una confianza determinada si por ejemplo en el año 2 existe una disminución representativa del caudal con respecto al año uno, se debe aplicar una prueba de comparación de muestras, cuyo desarrollo no es motivo de la presente obra.

En los diagramas de los años 1, 2 y 4, se puede apreciar que la mediana se encuentra aproximadamente en la mitad de las cajas; por consiguiente, se puede considerar que estas distribuciones son simétricas; a diferencia de la distribución de los datos del año 3 que indica un sesgo hacia la izquierda.

En el caso de requerir elaborar el diagrama de cajas a partir de una tabla de datos se puede agrupar las muestras en un *DataFrame* como se indica a continuación:

Código:

```
# Agrupación de las muestras en un DataFrame a partir del diccionario datos
datos = {'Año 1':muestra1, 'Año 2':muestra2, 'Año 3':muestra3, 'Año 4':muestra4}
muestras = pd.DataFrame(datos)
```

7.3 Diagrama de violín

El diagrama de violín es la combinación del histograma con el diagrama de caja, por lo que su aplicación es la suma de estas dos herramientas gráficas. Su elaboración se puede realizar mediante las librerías *seaborn* y *matplotlib.pyplot*.

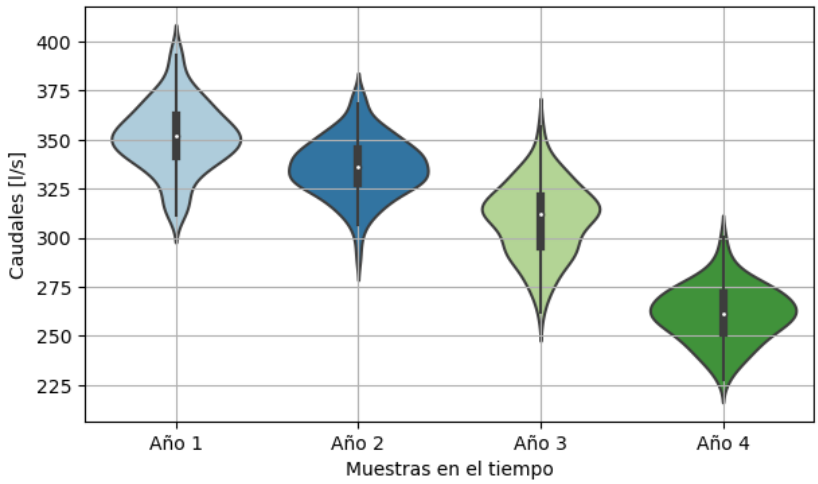
Código:

```
# Importación de las librerías
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Creación de 4 muestras aleatorias con la semilla 10
np.random.seed(10) # semilla = 10
# Creación de 4 muestras aleatorias con la semilla 10
np.random.seed(10) # semilla = 10
muestra1 = np.random.normal(350, 18, 100)
muestra2 = np.random.normal(335, 15, 100)
muestra3 = np.random.normal(310, 20, 100)
muestra4 = np.random.normal(260, 15, 100)

# Agrupación de las muestras en un DataFrame a partir del diccionario datos
datos = {'Año 1':muestra1, 'Año 2':muestra2, 'Año 3':muestra3, 'Año
4':muestra4}
muestras = pd.DataFrame(datos)

# Elaboración del diagrama de violín para comparar las muestras
plt.figure(figsize=(7, 4))
sns.violinplot(muestras, vert = True, palette='Paired')
plt.xlabel('Muestras en el tiempo')
plt.ylabel('Caudales [l/s]')
plt.grid(True)
```

Salida:

Figura 7.7

Comparación de 4 muestras mediante un diagrama de violines.



Nota. Los autores**.**

La librería *plotly.express*, proporciona una alternativa para elaborar un diagrama de violín dinámico, que proporciona herramientas como guardar, zoom, seleccionar, entre otras.

Código:

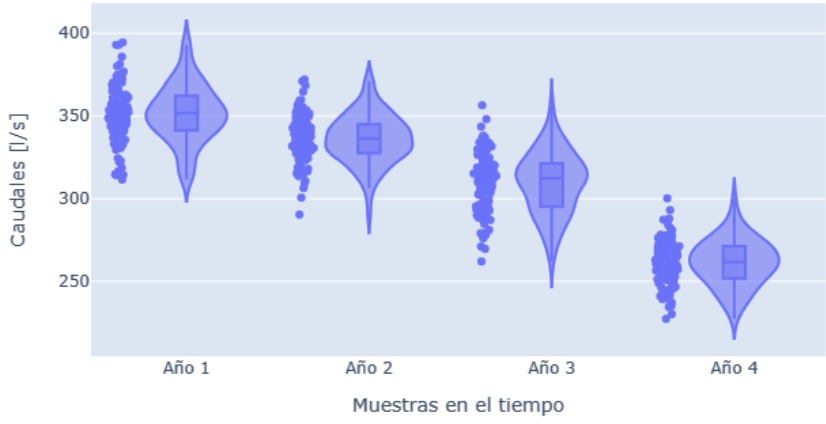
```
# Importación de las librerías
import plotly.express as px
import matplotlib.pyplot as plt
import numpy as np

# Creación de 4 muestras aleatorias con la semilla 10
np.random.seed(10) # semilla = 10
muestra1 = np.random.normal(350, 18, 100)
```

```
muestra2 = np.random.normal(335, 15, 100)
muestra3 = np.random.normal(310, 20, 100)
muestra4 = np.random.normal(260, 15, 100)

# Agrupación de las muestras en un DataFrame
datos = {'Año 1':muestra1, 'Año 2':muestra2, 'Año 3':muestra3, 'Año 4':muestra4}
muestras = pd.DataFrame(datos)

# Elaboración del diagrama de violín para comparar las muestras
fig = px.violin(muestras, box = True, points="all")
fig.update_layout(xaxis_title='Muestras en el tiempo',
                  yaxis_title='Caudales [l/s]',
                  width=700, height=400)
```

Salida:

Figura 7.8

Diagrama de violines con la librería plotly.express.



Nota. Los autores**.**

7.4 Gráficos de dispersión

Los diagramas de dispersión son herramientas poderosas que ayudan a visualizar datos, identificar tendencias, detectar anomalías y comprender las relaciones entre variables. Las librerías más utilizadas son *matplotlib.pyplot* y *seaborn* (Coder, 2023).

Código:

Opción 1:

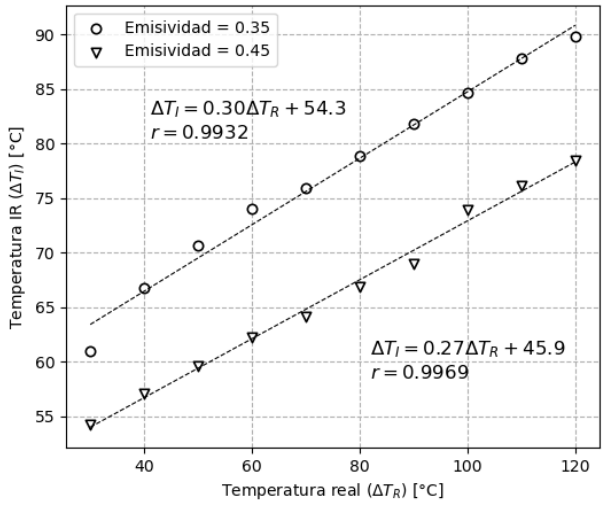
```
# Importación de las librerías
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt

# Configuración de las variables
T = [30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
E035 = [61, 66.8, 70.6, 74, 75.9, 78.8, 81.8, 84.6, 87.8, 89.8]
E045 = [54.2, 57.1, 59.6, 62.2, 64.1, 66.9, 69, 73.9, 76.1, 78.4]
X = pd.DataFrame({'T':T})

# Cálculo de los parámetros de la regresión lineal
pend, inter, r_val, p_val, std_err = stats.linregress(T, E035)
pend2, inter2, r_val2, p_val2, std_err2 = stats.linregress(T, E045)

# Variables dependientes de la regresión lineal
y1 = pend * X['T'] + inter
y2 = pend2 * X['T'] + inter2

# Elaboración del gráfico de dispersión con las líneas de tendencia
plt.rcParams.update({'figure.figsize':(6,5), 'figure.dpi':100})
fig, ax = plt.subplots()
ax.plot(T, y1, linestyle = "--", linewidth = 0.8, color = "k")
ax.plot(T, y2, linestyle = "--", linewidth = 0.8, color = "k")
ax.scatter(T, E035, label='Emisividad = 0.35', marker = "o", c = "white",
```

```
        edgecolors = "black", linewidths = 1.2)
ax.scatter(T, E045, label='Emisividad = 0.45', marker = "v", c = "white",
        edgecolors = "black", linewidths = 1.2)
ax.set_xlabel(r'Temperatura real ($\Delta$$T_R$) [°C]')
ax.set_ylabel(r'Temperatura IR ($\Delta$$T_I$) [°C]')

ax.text(41, 80.5,
      r'$\Delta T_I={0:.1f}\Delta T_R+{1:.1f}$'.format(pend,inter)+
      '\n'+r'$r={0:.1f}$'.format(r_value), fontsize=12)
ax.text(82, 58.5,
      r'$\Delta T_I={0:.1f}\Delta T_R+{1:.1f}$'.format(pend2,inter2)+
      '\n'+r'$r={0:.1f}$'.format(r_value2), fontsize=12)

ax.grid(linestyle = "dashed")
ax.set_axisbelow(True)
ax.legend()
plt.show()
```

Salida:

Figura 7.9

Diagrama de dispersión con la librería matplotlib.



Nota. Los autores**.**

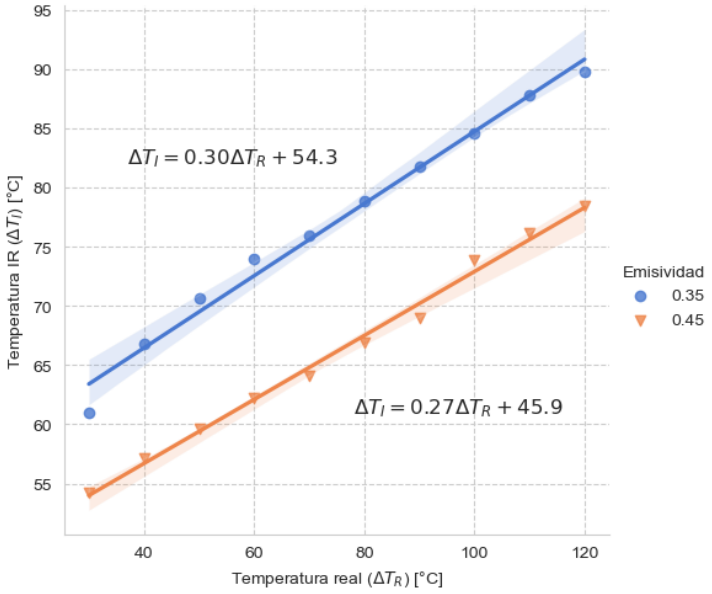
Opción 1:

```
# Importación de las librerías
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import seaborn as sns

T = [30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
E035 = [61, 66.8, 70.6, 74, 75.9, 78.8, 81.8, 84.6, 87.8, 89.8]
E045 = [54.2, 57.1, 59.6, 62.2, 64.1, 66.9, 69, 73.9, 76.1, 78.4]

# Cálculo de los parámetros de la regresión lineal
pend, inter, r_val, p_val, std_err = stats.linregress(T, E035)
pend2, inter2, r_val2, p_val2, std_err2 = stats.linregress(T, E045)

# Agrupación de las muestras en un DataFrame
n = len(T)
datos = pd.DataFrame({'T':T+T, 'E035':E035+E045,
                      'Emisividad':['0.35']*n + ['0.45']*n})

# Elaboración del gráfico de dispersión con lmplot
sns.set_style("whitegrid", {'grid.linestyle': '--'})
g = sns.lmplot(data=datos, x="T", y="E035", hue="Emisividad",
          markers=['o', 'v'], legend=True, fit_reg=True,
          palette = "muted", ci = 98) # Intervalo de confianza al 98%
g.fig.set_size_inches(6, 5)
plt.xlabel(r'Temperatura real ($\Delta$$T_R$) [°C]')
plt.ylabel(r'Temperatura IR ($\Delta$$T_I$) [°C]')
plt.text(37, 82,
        r'$\Delta T_I={0:.1f}\Delta T_R+{1:.1f}$'.format(pend,inter),
        fontsize=12)
plt.text(82, 62,
        r'$\Delta T_I={0:.1f}\Delta T_R+{1:.1f}$'.format(pend2,inter2),
        fontsize=12)
plt.show()
```

Salida:

Figura 7.10

Diagrama de dispersión con la librería seaborn.



Nota. Los autores**.**

7.4.1 Linealización y regresión lineal de la función de Weibull

Para profundizar en la aplicación de los diagramas de dispersión se desarrolla un ejemplo en el que se encuentra los parámetros de la función conocida como Weibull (Ecuación (35)) a la que se ajusta los tiempos hasta la falla del retenedor de un pistón neumático en horas (TTF = [2445, 4535, 6800, 10520, 16500]).

$$F(t) = 1 - e^{-\left(\frac{t}{\eta}\right)^{\beta}}, \tag{35}$$

donde $F(t)$ es la probabilidad de que el retenedor (el elemento de estudio) falle desde que es nuevo hasta el tiempo t (En otras palabras, es la probabilidad acumulativa de que el evento ocurra), e es el número de Euler que es de 2,7182818284..., t es el tiempo que transcurre hasta que ocurre la falla (evento estudiado), β es el parámetro de forma de la distribución de Weibull y η es el parámetro de escala de la distribución de Weibull.

Código para la presentación de los datos:

```
# Importación de la librería
import numpy as np
import pandas as pd

# Presentación de los datos
TTF = [2445, 4535, 6800, 10520, 16500]
n = len(TTF)
i = list(range(1, n + 1))
datos = pd.DataFrame({'i':i,'TTF':TTF})
datos['F(t)'] = (datos['i'] - 0.3) / (n + 0.4)

# Linealización de los datos
datos['X'] = np.log(datos['TTF'])
datos['y'] = np.log(np.log(1/(1-datos['F(t)'])))
print(datos)
```

Salida:

	i	TTF	F(t)	X	y
0	1	2445	0.129630	7.801800	-1.974459
1	2	4535	0.314815	8.419580	-0.972686
2	3	6800	0.500000	8.824678	-0.366513
3	4	10520	0.685185	9.261033	0.144767
4	5	16500	0.870370	9.711116	0.714455

Código para el cálculo de los parámetros de Weibull mediante el método de la linealización:

```
params = np.polyfit(datos['X'], datos['y'], 1)
r_val = np.corrcoef(datos['X'], datos['y'])[0, 1]
b0, beta = params[1], params[0]
eta = np.exp(-b0/beta)
print('r: %.4f' % r_val)
print('\u03B2: %.2f' % beta)
print('\u03B7: %.0f' % eta)
```

Salida:

r: 0.9973
β: 1.40
η: 9454
Código:

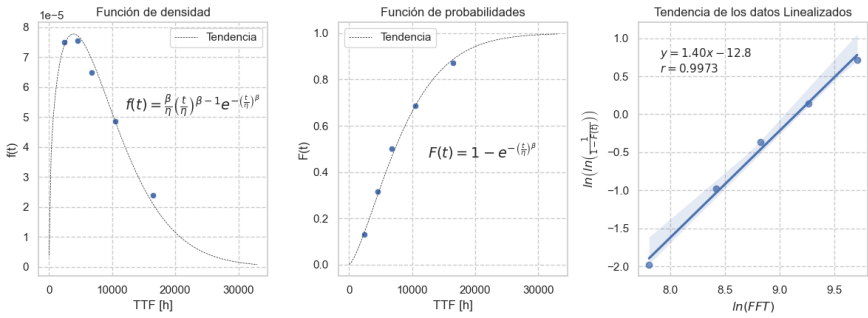
```
# Importación de la librería
import matplotlib.pyplot as plt
import seaborn as sns

x_2 = np.linspace(1, datos['TTF'].max()*2, num=100)
y_2 = beta/eta*(x_2/eta)**(beta-1)*np.exp(-(x_2/eta)**beta)
y_3 = 1-np.exp(-(x_2/eta)**beta)
datos['f(t)']=beta/eta*(datos['TTF']/eta)**(beta-1)*(1-datos['F(t)'])

# Elaboración de los diagramas de dispersión
sns.set_style("whitegrid", {'grid.linestyle': '--'}) # Configurar el estilo de Seaborn
fig, axes = plt.subplots(1, 3, figsize=(12, 4.5)) # Crear 2 filas y 2 columnas

sns.scatterplot(data=datos, x='TTF', y='f(t)', ax=axes[0])
axes[0].plot(x_2, y_2, linewidth=0.5, linestyle = "--", color = "k", label='Tendencia')
axes[0].text(12000, 5.2e-5,r'$f(t)=\frac{\beta}{\eta}\left(\frac{t}{\eta}\right)^{\beta-1}e^{-\left(\frac{t}{\eta}\right)^{\beta}}$', fontsize=14)
axes[0].set_title("Función de densidad")
axes[0].set_xlabel('TTF [h]')
axes[0].set_ylabel('f(t)')
axes[0].legend();

sns.scatterplot(data=datos, x='TTF', y='F(t)', ax=axes[1])
axes[1].plot(x_2, y_3, linewidth=0.5, linestyle = "--", color = "k", label='Tendencia')
axes[1].text(12500, 0.46,r'$F(t)=1-e^{-\left(\frac{t}{\eta}\right)^{\beta}}$', fontsize=14)
axes[1].set_title("Función de probabilidades")
axes[1].set_xlabel('TTF [h]')
axes[1].set_ylabel('F(t)')
axes[1].legend();

sns.regplot(data=datos, x="X", y="y", ax=axes[2])
axes[2].text(7.9, 0.55,
```

```
        r'$y={0:.2f}x{1:.1f}$'.format(beta,b0)+
        '\n'r'$r={0:.4f}$'.format(r_val), fontsize=11)
axes[2].set_title("Tendencia de los datos Linealizados")
axes[2].set_xlabel(r'$ln\left(FFT\right)$')
axes[2].set_ylabel(r'$ln\left(ln\left(\frac{1}{1-F(t)}\right)\right)$')

plt.tight_layout()
plt.show()
```

Salida:

Figura 7.11

Diagrama de dispersión con la librería seaborn.



Nota. Los autores**.**

Si se reemplazan los parámetros encontrados en la Ecuación (35) y se despeja t, se obtiene la Ecuación (36); utilizada para encontrar el rango de tiempo en el que es improbable que el retenedor falle ($F(t) = 0{,}05$) que es de $1133\ horas$.

$$t = 9454\ h \left(ln\left(\frac{1}{1 - F(t)}\right)\right)^{\frac{1}{1,4}}, \quad (36)$$

Al comparar este tiempo ($1133\ h$) con la vida característica $\eta = 9454$ h, se puede notar de que existe mucha diferencia entre estos dos parámetros, ratificándose la elevada dispersión de los datos que sugiere el valor de 1,4 del parámetro de forma β, ya que es muy cercano a 1. Por lo tanto, se concluye que para este elemento no es técnicamente factible la sustitución a intervalos fijos de tiempo (mantenimiento predeterminado) en tanto que la inspección directa de retenedor no es posible ya que cuando un pistón se desarma, obligatoriamente se debe cambiar los retenedores. Ante esto, la estrategia más eficaz es monitorear el deterioro del retenedor mediante la detección de disminuciones de la presión de entrada.

7.5 Diagramas de barras

Los diagramas de barras son muy utilizados para representar valores absolutos o relativos. A diferencia de los histogramas, se pueden utilizar para comparar valores categóricos y las barras son más delgadas y separadas.

En el siguiente diagrama de barras, se representan las medias con sus respectivos errores estándar, de los niveles de vibración global rms en mm/s de 4 máquinas, donde una de ellas no tiene datos (datos faltantes). La máquina con datos faltantes no se puede eliminar debido a que existe físicamente.

Estas mediciones fueron tomadas previo a la alineación de sus ejes, con la finalidad de que posteriormente se realice una comparación del nivel de vibración mediante diagramas de barras.

Código:

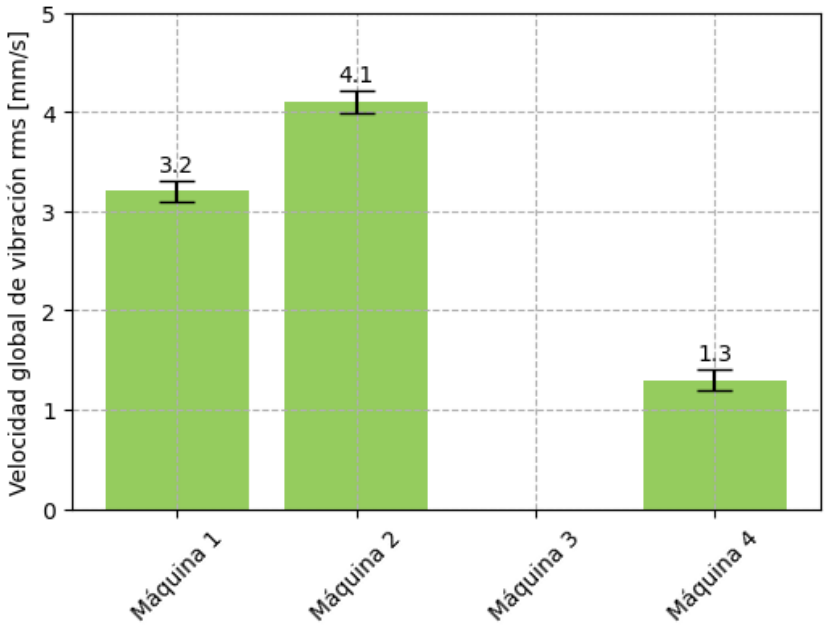
```
# Importación de las librerías
import matplotlib.pyplot as plt
import numpy as np

# Presentación de los datos
maquinas = ["Máquina 1", "Máquina 2", "Máquina 3","Máquina 4"]
vibracion = [3.2, 4.1, np.NaN, 1.3]
error_std = [0.1, 0.12, np.NaN, 0.1]

# Elaboración del diagrama de barras
plt.rcParams.update({'figure.figsize':(6, 4), 'figure.dpi':100})
plt.ylim(0, 5)

plt.bar(maquinas, vibracion, yerr = error_std, capsize = 8,
        color = "#95CC5E", alpha = 1)
```

```
for i, k in enumerate(vibracion):
    plt.text(i, k+error_std[i]/2+0.1, str(k), ha='center',
             va='bottom', rotation=0)

plt.xticks(rotation=45)
plt.ylim(bottom=0)
plt.ylabel('Velocidad global de vibración rms [mm/s]')
plt.grid(linestyle = "dashed")
plt.show()
```

Salida:

Figura 7.12

Diagrama de barras con matplotlib.



Nota. Los autores**.**

Código:

Opción 1:

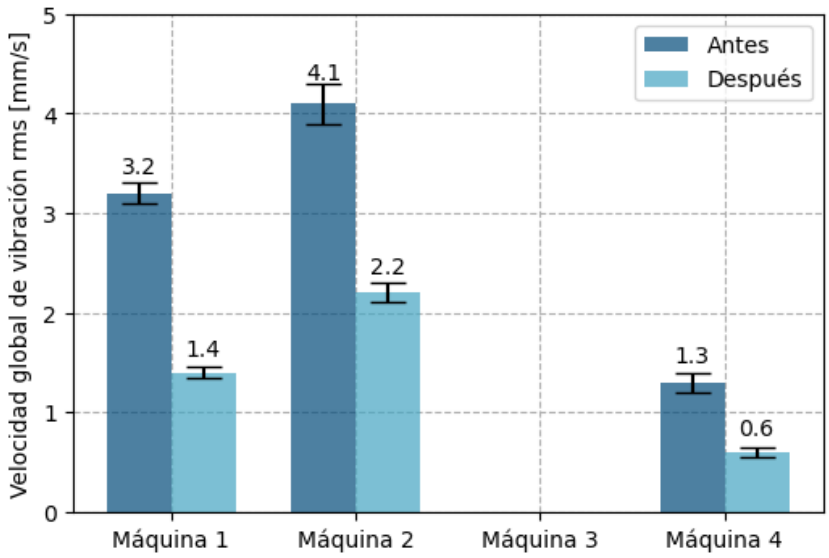
```
# Importación de las librerías
import matplotlib.pyplot as plt
import numpy as np

# Presentación de los datos
maquina = [1, 2, 3, 4]
vibracion1 = [3.2, 4.1, np.NaN, 1.3]
vibracion2 = [1.4, 2.2, np.NaN, 0.6]
error_std1 = [0.1, 0.2, np.NaN, 0.1]
error_std2 = [0.05, 0.1, np.NaN, 0.05]
maquinas = ["Máquina 1", "Máquina 2", "Máquina 3","Máquina 4"]

# Elaboración del diagrama de barras
ancho_barra = 0.35 # Ancho de las barras
x = np.arange(len(maquinas)) # Rango de valores en el eje x
plt.figure(figsize=(6, 4))  # Tamaño de la figura (ancho, alto)

fig, ax = plt.subplots()
ax.set_ylim(0, 5) # Rango del eje y

ax.bar(x - ancho_barra/2, vibracion1, ancho_barra, yerr = error_std1,
       capsize = 8, color = "#024b7a", alpha = 0.7, label='Antes')
for i, k in enumerate(vibracion1):
    ax.text(i-ancho_barra/2, k+error_std1[i]/2+0.1, str(k),
            ha='center', va='bottom', rotation=0)

ax.bar(x + ancho_barra/2, vibracion2, ancho_barra, yerr = error_std2,
       capsize = 8, color = "#44a5c2", alpha = 0.7, label='Después')
for i, k in enumerate(vibracion2):
    ax.text(i+ancho_barra/2, k+error_std2[i]/2+0.1, str(k),
            ha='center', va='bottom', rotation=0)
```

```
# Configuración adicional (etiquetas, título, leyenda, etc.)
ax.set_ylabel('Velocidad global de vibración rms [mm/s]')
ax.set_xticks(x, maquinas)
ax.set_xticklabels(maquinas, rotation= 0)
ax.grid(linestyle = "dashed")
ax.set_axisbelow(True)
ax.legend()
plt.show()
```

Salida:

Figura 7.13

Diagrama de barras comparativas elaborado con matplotlib.



Nota. Los autores**.**

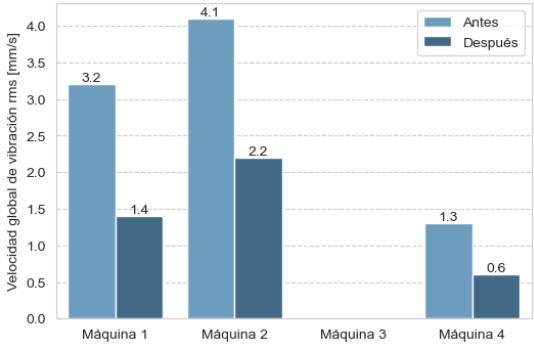
Opción 2:

```
# Importación de las librerías
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Presentación de los datos
vibracion1 = [3.2, 4.1, np.NaN, 1.3]
vibracion2 = [1.4, 2.2, np.NaN, 0.6]
vibracion = vibracion1 + vibracion2
maquinas = ["Máquina 1", "Máquina 2", "Máquina 3","Máquina 4"]
maquinas = maquinas + maquinas
estado = ['Antes']*4 + ['Después']*4

# Elaboración del diagrama de barras
plt.figure(figsize=(6, 4))  # Tamaño de la figura (ancho, alto)
sns.set_style("whitegrid", {'grid.linestyle': '--'})
sns.barplot(x = maquinas, y = vibracion, hue = estado,
            palette = "Blues_d")
for i, k in enumerate(vibracion1):
    plt.annotate(str(k), xy=(i-0.2, k), ha='center', va='bottom')
for i, k in enumerate(vibracion2):
    plt.annotate(str(k), xy=(i+0.2, k), ha='center', va='bottom')
plt.ylabel('Velocidad global de vibración rms [mm/s]')
```

Salida:

Figura 7.14

Diagrama de barras comparativas elaborado con seaborn.



Nota. Los autores**.**

Los diagramas de barras de las Figuras 7.13 y 7.14 son suficientes para representar las diferencias en el nivel de vibración del antes y después de la alineación de sus ejes; sin embargo, a continuación, se presenta otra alternativa con barras verticales sucesivas.

Código:

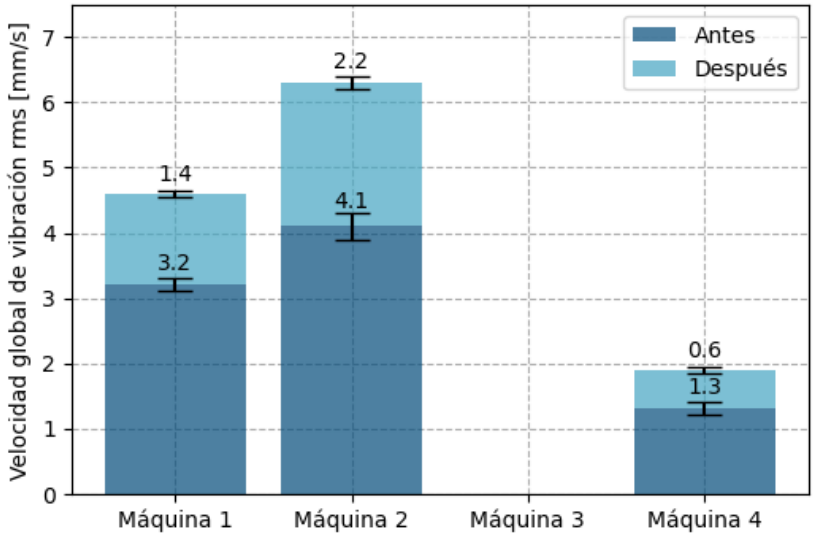
```
# Importación de las librerías
import matplotlib.pyplot as plt
import numpy as np

# Presentación de los datos
maquina = [0, 1, 2, 3]
vibracion1 = [3.2, 4.1, np.NaN, 1.3]
vibracion2 = [1.4, 2.2, np.NaN, 0.6]
error_std1 = [0.1, 0.2, np.NaN, 0.1]
error_std2 = [0.05, 0.1, np.NaN, 0.05]
maquinas = ["Máquina 1", "Máquina 2", "Máquina 3","Máquina 4"]

# Elaboración del diagrama de barras
plt.figure(figsize=(6, 4))  # Tamaño de la figura (ancho, alto)
fig, ax = plt.subplots()
ax.set_ylim(0, 7.5) # Rango del eje y

ax.bar(maquina, vibracion1, yerr = error_std1, capsize = 8,
      tick_label = maquinas, color = "#024b7a", alpha = 0.7, label='Antes')
for i, k in enumerate(vibracion1):
   ax.text(i, k+error_std1[i]/2+0.1, str(k),
         ha='center', va='bottom', rotation=0)

ax.bar(maquina, vibracion2, bottom = vibracion1, yerr = error_std2,
      capsize = 8, tick_label = maquinas, color = "#44a5c2",
      alpha = 0.7, label='Después')
```

```
for i, k in enumerate(vibracion2):
    ax.text(i, k+vibracion1[i]+error_std2[i]/2+0.1, str(k),
            ha='center', va='bottom', rotation=0)

# Configuración adicional (etiquetas, título, leyenda, etc.)
ax.set_ylabel('Velocidad global de vibración rms [mm/s]')
ax.set_xticklabels(maquinas, rotation=0)
ax.grid(linestyle = "dashed")
ax.set_axisbelow(True)
ax.legend()
plt.show()
```

Salida:

Figura 7.15

Diagrama de barras vertical del antes y después.



Nota. Los autores**.**

7.6 Diagramas de pastel

Un diagrama de pastel es una representación gráfica utilizada para mostrar la proporción de un conjunto de datos categóricos en relación con el todo, para conocer la contribución de cada categoría al total; sin embargo, es importante reconocer que cuando se trata de datos con muchas categorías o cuando se necesita una representación más precisa, se debe utilizar otros gráficos como el diagrama de barras que se trató en el subtema anterior.

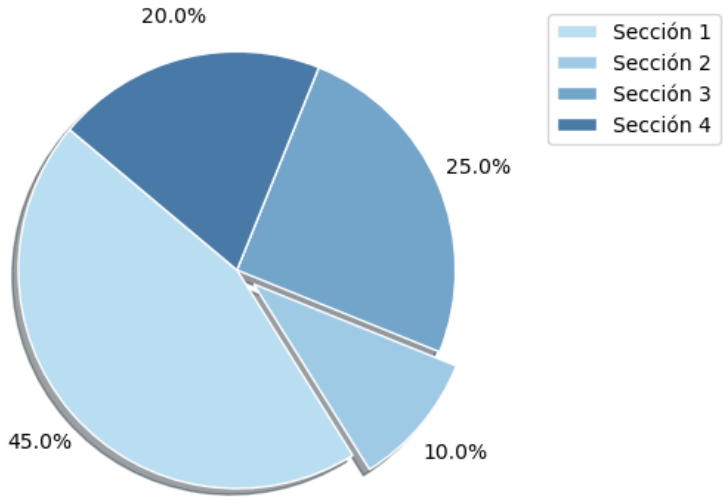
Código:

```
# Importación de la librería
import matplotlib.pyplot as plt

# Presentación de los datos
cantidad = [9, 2, 5, 4]
n_bombas = ["Sección 1", "Sección 2", "Sección 3", "Sección 4"]
explode = [0, 0.1, 0, 0]
colores = ["#B9DDF1", "#9FCAE6", "#73A4CA", "#497AA7", "#2E5B88"]

# Elaboración del diagrama de pastel
plt.figure(figsize=(10, 4))  # Tamaño de la figura (ancho, alto)
fig, ax = plt.subplots()
ax.pie(cantidad, autopct = '%1.1f%%', pctdistance = 1.2, startangle=140,
       shadow = True, explode = explode, colors = colores,
       wedgeprops = {"linewidth": 1, "edgecolor": "white"})

ax.legend(n_bombas, loc='center left', bbox_to_anchor=(1.05, 0.85))
plt.show()
```

Salida:

Figura 7.16

Diagrama de pastel elaborado con matplotlib.



Nota. Los autores**.**

8

Capítulo 8

Casos prácticos

Capítulo 8

8. Casos prácticos

Una vez que se han explorado los métodos más comunes aplicados en el cálculo de los parámetros y medidas del análisis descriptivo de datos, es muy oportuno realizar el desarrollo de un caso de estudio para utilizar otras alternativas de funciones de *Python*, que ofrecen una solución más compacta y clara para el análisis exploratorio de varias muestras asociadas, que suele comúnmente requerirse en la gestión del mantenimiento industrial y el monitoreo de la condición de máquinas e instalaciones.

8.1 Planteamiento del problema

Un fabricante de bombas centrífugas para la minería a cielo abierto realizó un estudio sobre la disminución del caudal con el transcurso del tiempo de operación bajo un determinado contexto operacional caracterizado por un fluido muy abrasivo.

En este estudio se recolectaron 4 muestras del caudal cada 2 años partiendo del año cero donde la bomba alcanzó los 350 l/s promedio (función inherente).

Durante el estudio se ha tomado la precaución de realizar actividades de mantenimiento rutinarias a la bomba, como limpieza de la succión y descarga, alineación y relubricaciones; por lo que se garantiza que la disminución del caudal es causada exclusivamente por el desgaste del impulsor.

Un cliente que plantea utilizar una de estas bombas en un contexto operacional similar y aplicando las mismas estrategias de mantenimiento empleadas en el estudio realizado, desea conocer el tiempo aproximado en el que el caudal disminuye hasta 150 l/s. Este valor marca el momento en el que la bomba experimenta una falla funcional y ya no puede cumplir con la función requerida.

Con estas condiciones el fabricante sabe que debe realizar un análisis descriptivo de los datos de la bomba y posteriormente una regresión lineal, para lo cual se sigue el siguiente desarrollo.

8.2 Desarrollo

El primer paso en el desarrollo es cargar las librerías de *Python* requeridas para la aplicación de las funciones que se van a emplear, dentro de las cuales se encuentran *NumPy*, *Pandas*, *SciPy*, *Sklearn*, *Matplotlib* y *Seaborn*.

Código:

```
# Importación de librerías
import numpy as np
import pandas as pd
import scipy.stats as stats
from sklearn.impute import KNNImputer
import matplotlib.pyplot as plt
import seaborn as sns
```

Seguidamente se realiza la importación de los datos desde la Hoja 1 del archivo de Excel que se puede descargar del siguiente enlace: https://acortar.link/vEeWcM.

Código:

```
# Importación de la Hoja1 del archivo de Excel
caudales0 = pd.read_excel('BA01.xlsx', sheet_name='Hoja1')
# Mostrar la información general de tabla_caudales
print(caudales0.info())
# Mostrar las primeras filas, por defecto muestra las 5 primeras filas
print(caudales0.head())
```

Salida:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0       n         500 non-null    int64
 1   caudal_1  496 non-null    float64
 2   caudal_2  496 non-null    float64
 3   caudal_3  496 non-null    float64
 4   caudal_4  495 non-null    float64
dtypes: float64(4), int64(1)
memory usage: 19.7 KB
None
```

	N	caudal_1	caudal_2	caudal_3	caudal_4
0	1	353.4	108.3	37.9	9.3
1	2	377.7	94.9	32.9	4.9
2	3	349.3	94.8	34.7	NaN
3	4	347.5	80.4	34.4	3.7
4	5	336.8	85.1	33.0	5.0

La primera exploración de los datos, indica que se cuenta con una tabla de 5 columnas y 500 registros de los cuales, la primera es una enumeración del 1 al 500, mientras que las otras corresponden a las mediciones de caudales en las que existen datos faltantes, por lo que se procede a realizar una imputación mediante el método de los k-vecinos, para reemplazar estos valores nulos por el promedio de sus 5 vecinos; pero previamente se elimina la primera columna, ya que no se utiliza.

Código:

```
del caudales0['n'] # Elimina la columna n
# Imputación de datos faltantes k-vecinos
imputer = KNNImputer(n_neighbors = 5) # se toma 5 vecinos
caudales1 = imputer.fit_transform(caudales0)
caudales1 = pd.DataFrame(caudales1, columns=caudales0.columns)
print(caudales1.info())
```

Salida:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   caudal_1  500 non-null    float64
 1   caudal_2  500 non-null    float64
 2   caudal_3  500 non-null    float64
 3   caudal_4  500 non-null    float64
dtypes: float64(4)
memory usage: 15.8 KB
None
```

Luego de imputar los datos faltantes, se obtuvo 500 valores no nulos del tipo flotante en cada una de las 4 columnas. En el siguiente paso se identifica y reemplaza los valores atípicos por la mediana.

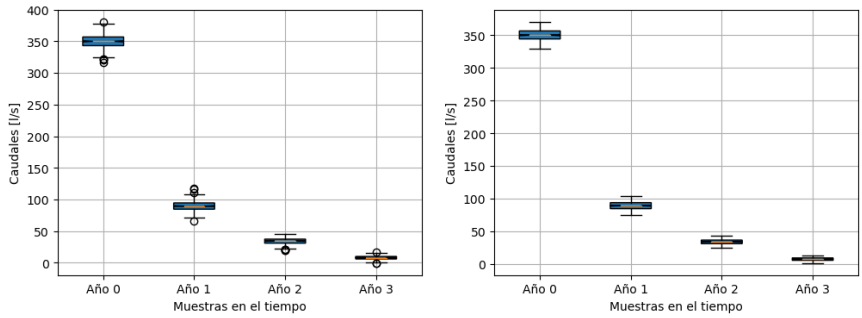
Código:

```
caudales = caudales1.copy() # Crea una copia de los datos
def tratar_valores_atipicos(columna):
    mediana = columna.median()
    umbral = 1.5 * (columna.quantile(0.75) - columna.quantile(0.25))
    valores_atipicos = (columna > mediana + umbral) | (columna < mediana
- umbral)
    columna[valores_atipicos] = mediana
# Aplicar la función para tratar valores atípicos a cada columna del
DataFrame
for columna in caudales.columns:
    tratar_valores_atipicos(caudales[columna])
fig, axs = plt.subplots(1, 2, figsize=(12, 4))  # 1 fila, 2 columnas
# Diagrama de cajas para visualizar los atípicos
axs[0].boxplot(caudales1, labels=['Año 0', 'Año 1', 'Año 2', 'Año 3'],
               vert=True, patch_artist=True, notch ='False')
axs[0].set_xlabel('Muestras en el tiempo')
axs[0].set_ylabel('Caudales [l/s]')
axs[0].grid(True)
# Diagrama de cajas para visualizar los datos sin valores atípicos
axs[1].boxplot(caudales, labels=['Año 0', 'Año 1', 'Año 2', 'Año 3'],
           vert=True, patch_artist=True, notch ='True')
axs[1].set_xlabel('Muestras en el tiempo')
axs[1].set_ylabel('Caudales [l/s]')
axs[1].grid(True)
plt.show()

print('Con atípicos:\n',caudales1.describe(),'\n')
print('Sin atípicos:\n',caudales.describe())
```

Salida:

Figura 8.1

Diagramas de caja.



Nota. Los autores**.**

Con atípicos:

	caudal_1	caudal_2	caudal_3	caudal_4
count	500.000000	500.000000	500.000000	500.000000
mean	350.226760	89.718920	34.221800	8.062360
std	10.096962	7.309128	4.657115	2.959412
min	317.100000	66.400000	19.800000	-0.400000
25%	343.200000	84.700000	30.975000	5.975000
50%	350.050000	89.600000	34.300000	8.000000
75%	357.000000	94.625000	37.400000	10.025000
max	381.000000	117.200000	46.100000	17.100000

Sin atípicos:

	caudal_1	caudal_2	caudal_3	caudal_4
count	500.000000	500.000000	500.000000	500.00000
mean	350.278060	89.713920	34.248800	8.02616
std	8.759491	6.420955	4.002816	2.59142
min	329.400000	74.900000	24.900000	2.00000
25%	344.200000	85.000000	31.500000	6.10000
50%	350.050000	89.600000	34.300000	8.00000
75%	356.550000	94.225000	37.100000	9.80000
max	370.500000	104.200000	43.300000	14.00000

Los diagramas de caja de la Figura 8.1 revelan la presencia inicial de valores atípicos en las 4 muestras, mismos que fueron reemplazados por su respectiva mediana; por lo que la función *describe*() muestra ligeros cambios en las medidas descriptivas de la muestra como la media, mediana (50%), desviación estándar (*std*), entre otras.

Código:

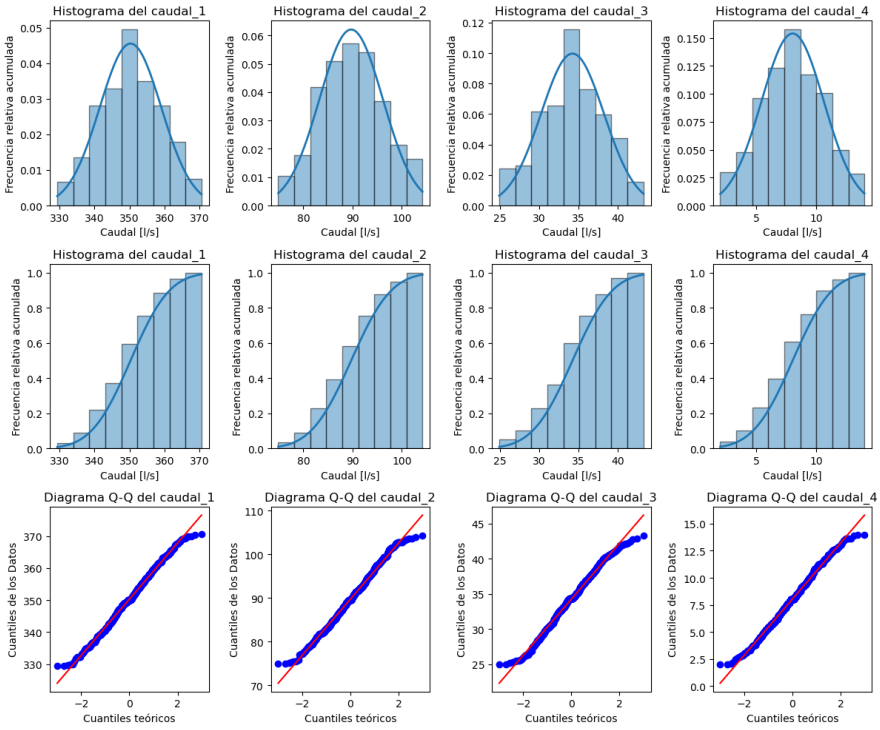
```
q = 9
fig, axs = plt.subplots(3, 4, figsize=(12, 10))  # 3 filas, 4 columnas

# Crear histograma para cada columna y agregarlo a los subplots
for i, columna in enumerate(caudales.columns):
    caudal = caudales[columna]
    x_1 = np.linspace(caudal.min(), caudal.max(), num=100)
    y_1 = stats.norm.pdf(x_1, caudal.mean(), caudal.std(ddof=0))
    axs[0,i].plot(x_1, y_1, linewidth=2)
     axs[0,i].hist(caudal, density=True, bins=q, color="#3182bd", alpha=0.5,
ec="black")
    axs[0,i].set_title(f'Histograma del {columna}')
    axs[0,i].set_xlabel('Caudal [l/s]')
    axs[0,i].set_ylabel('Frecuencia relativa acumulada')

    y_2 = stats.norm.cdf(x_1, caudal.mean(), caudal.std(ddof=0))
    axs[1,i].plot(x_1, y_2, linewidth=2)
     axs[1,i].hist(caudal, density=True, bins=q, color="#3182bd", alpha=0.5,
ec="black", cumulative=True)
    axs[1,i].set_title(f'Histograma del {columna}')
    axs[1,i].set_xlabel('Caudal [l/s]')
    axs[1,i].set_ylabel('Frecuencia relativa acumulada')

    stats.probplot(caudal, dist='norm', plot=axs[2,i])
```

```
    axs[2,i].set_title(f'Diagrama Q-Q del {columna}')
    axs[2,i].set_xlabel('Cuantiles teóricos')
    axs[2,i].set_ylabel('Cuantiles de los Datos')


plt.tight_layout()  # Ajustar el espaciado entre subplots
plt.show()
```

Salida:

Figura 8.2

Descripción gráfica de las muestras.



Nota. Los autores**.**

En la Figura 8.2, se puede observar que los histogramas de las muestras tienden a ajustarse a la campana de Gauss, distribuyéndose de forma apreciablemente simétrica; sin embargo, los diagramas Q-Q indican no coincidencia de los puntos con la recta en los extremos, lo que sugiere que los datos no se distribuyen normalmente. Para comprobar lo aseverado se procede a la aplicación de técnicas analíticas:

Código:

```
# Cálculo del coeficiente de simetría
simetria = caudales.skew()
print('Simetría de cada caudal:\n', simetria)
```

Salida:

```
Simetría de cada caudal:
caudal_1    -0.019305
caudal_2     0.048615
caudal_3    -0.136128
caudal_4     0.014760
dtype: float64
```

Como los valores del coeficiente de asimetría de las 4 muestras están entre 1 y -1 se revela que existe un ligero sesgo; en donde, las muestras 1 y 3 tienen sesgo hacia la izquierda y las muestras 2 y 4 hacia la derecha.

Código:

```
# Cálculo de la curtosis
curtosis = caudales.kurtosis()
print('Curtosis de cada caudal:\n', curtosis)
```

Salida:

```
Curtosis de cada caudal:
caudal_1   -0.475408
caudal_2   -0.509990
caudal_3   -0.501753
caudal_4   -0.530038
dtype: float64
```

Como la curtosis de las 4 muestras son menor que 3, se concluye que la dispersión de los datos de cada una de ellas es menor que la distribución normal (platicúrtica).

Código:

```
p_valor = caudales.apply(lambda x: stats.shapiro(x))
print('P valor:\n', p_valor.iloc[1])
```

Salida:

P valor:
caudal_1 0.028117
caudal_2 0.008269
caudal_3 0.001654
caudal_4 0.012590
Name: 1, dtype: float64

Como el p valor de las 4 muestras son menores a 0,05, se rechaza la hipótesis nula y se acepta la hipótesis alternativa, concluyendo que existe suficiente evidencia para aseverar que los datos no se distribuyen normalmente; por lo que es necesario normalizar; sin embargo, no se procede a hacerlo puesto que para la regresión lineal únicamente se utiliza las medias de cada muestra, sin que afecte la distribución de estos datos.

Código:

```
# Obtención de las medias de cada muestra

t = [0, 2, 4, 6] #
Q = [caudales[columna].mean() for columna in caudales.columns]
caudales2 = pd.DataFrame({'t':t, 'Q':Q })
print(caudales2)

# Diagrama de tendencias
```

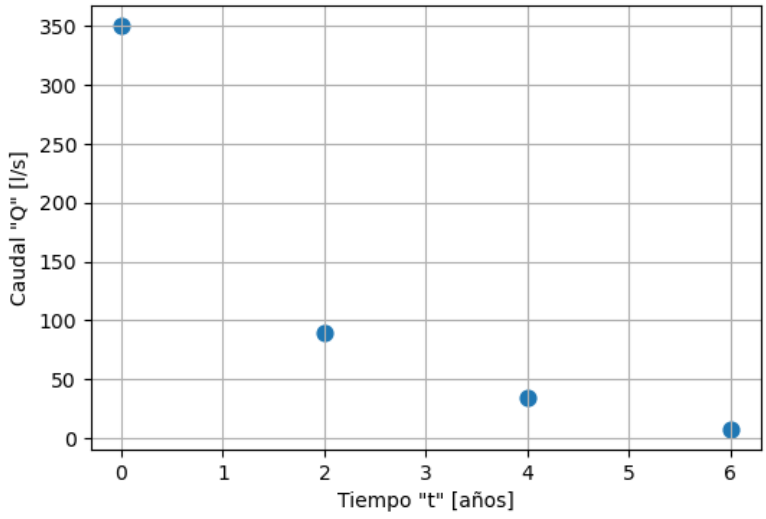
```
plt.figure(figsize=(6, 4))
plt.scatter(x=t, y=Q, s = 60)
plt.xlabel('Tiempo "t" [años]')
plt.ylabel('Caudal "Q" [l/s]')
plt.grid(True)
plt.show()
```

Salida:

	t	Q
0	0	350.27806
1	2	89.71392
2	4	34.24880
3	6	8.02616

Figura 8.3

Diagrama de pastel elaborado con *matplotlib*.



Nota. Los autores**.**

Como se indica en el diagrama de dispersión de las medias de las muestras de la Figura 8.3, el caudal disminuye con los años de forma no lineal; donde se observa que cuando $t = 0$, $Q = 350\ l/s$; esto es $Q(0) = 350\ l/s$. En este punto, el caudal es máximo, mismo que disminuye rápidamente, debido a que existe una elevada cantidad de abrasivo en el fluido. A medida que el impulsor se desgasta el caudal disminuye, y con ello también disminuye la cantidad de abrasivo; por lo que, la rapidez con que varían el desgaste del impulsor y el caudal se reduce a un ritmo constante, tendiendo a 0 en un tiempo prolongado ($\lim_{t\to\infty} Q(t) = 0$). Este comportamiento es congruente con datos que tienen una tendencia exponencial.

$$Q = b_0 e^{b_1 t}. \tag{37}$$

Aplicando logaritmos a la Ecuación (37), se obtiene:

$$\ln(Q) = \ln(b_0) + b_1 t, \tag{38}$$

donde $\ln(Q)$ es la variable dependiente de la función linealizada, $\ln(b_0)$ es la intersección (en la regresión lineal no se determina el valor de b_0 directamente) y b_1 es la pendiente. Por lo que reemplazando se obtiene:

$$\mathrm{Y} = \ln(Q),$$
$$B_0 = \ln(b_0),$$
$$b_1 = b_1,$$
$$\mathrm{X} = t.$$

Con lo que la Ecuación (37) se asemeja a la siguiente función lineal:

$$\mathrm{Y} = B_0 + b_1 X, \tag{39}$$

Código:

```
# Linealización de los datos
caudales2['ln(Q)'] = np.log(caudales2['Q'])
print(caudales2)
```

Salida:

	t	Q	ln(Q)
0	0	350.27806	5.858727
1	2	89.71392	4.496626
2	4	34.24880	3.533652
3	6	8.02616	2.082706

Una vez que se ha incrementado la columna de $ln(Q)$, fruto del proceso de linealización, se procede con la regresión lineal para encontrar los parámetros

Código:

```
b1, B0 = np.polyfit(caudales2['t'], caudales2['ln(Q)'], 1)
r_val = np.corrcoef(caudales2['t'], caudales2['ln(Q)'])[0, 1]
b0=  np.exp(B0)
print('r:           %.4f' % r_val)
print('Intersección: %.4f' % B0)
print('Pendiente:    %.4f' % b1)
print('b0 = e^B0:    %.4f' % b0)
```

Salida:

```
r:            -0.9973
Intersección:  5.8366
Pendiente:    -0.6146
b0 = e^B0:    342.6068
```

Como el coeficiente de correlación r de Pearson obtenido es $-0{,}9973 \approx -1$, se infiere que existe un buen ajuste de las variables linealizadas a una recta decreciente.

Código:

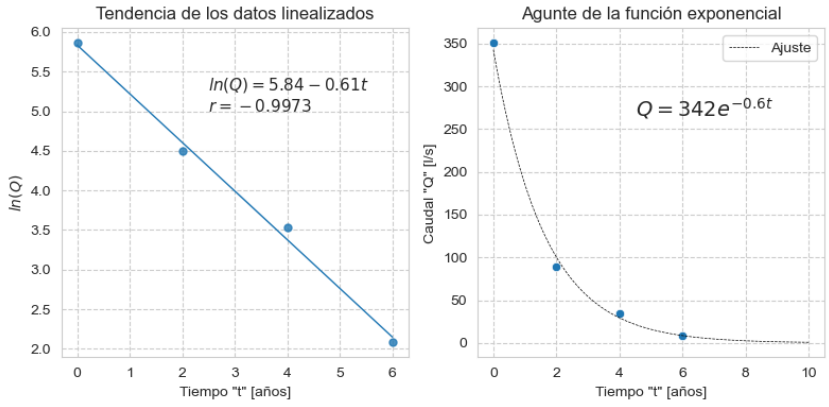
```
X = np.linspace(0, 10, num=100)
y = b0*np.exp(b1*X)

# Elaboración de los diagramas de dispersión
sns.set_style("whitegrid", {'grid.linestyle': '--'}) # Configurar el estilo de Seaborn
fig, axes = plt.subplots(1, 2, figsize=(8, 4)) # Crear 1 fila y 2 columnas

sns.regplot(data=caudales2, x="t", y="ln(Q)", ci=5,
            scatter_kws={"s": 25}, line_kws={'linewidth': 1}, ax=axes[0])
```

```
axes[0].text(2.5, 5,
        r'$ln(Q)={0:.2f}{1:.2f} t$'.format(B0,b1)+
        '\n'+r'$r={0:.4f}$'.format(r_val), fontsize=11)
axes[0].set_title("Tendencia de los datos linealizados")
axes[0].set_xlabel('Tiempo "t" [años]')
axes[0].set_ylabel(r'$ln(Q)$')

sns.scatterplot(data=caudales2, x='t', y='Q', ax=axes[1])
axes[1].plot(X, y, linewidth=0.5, linestyle = "--", color = "k", label='Ajuste')
axes[1].text(4.5, 265,r'$Q=342e^{-0.6t}$', fontsize=14)
axes[1].set_title("Agunte de la función exponencial")
axes[1].set_xlabel('Tiempo "t" [años]')
axes[1].set_ylabel('Caudal "Q" [l/s]')
axes[1].legend();

plt.tight_layout()
plt.show()
```

Salida:

Figura 8.4

Diagrama de la linealización y ajuste a la función exponencial.



Nota. Los autores**.**

La figura indica que existe un buen ajuste ya que la línea de tendencia pasa por todos los puntos; sin embargo, para verificar se plantea realizar la prueba de Pearson:

Código:

```
# Cálculo de correlación y significancia con Scipy
r, p = stats.pearsonr(caudales2['t'],caudales2['ln(Q)'])
print('Correlación r de Pearson:\n', 'r = %.4f' % r,
      'p valor = %.8f' % p)
```

Salida:

Correlación r de Pearson:
r = -0.9973 p valor = 0.00272402

Como el $p\,valor = 0{,}00272402 < 0{,}05$, se rechaza la hipótesis nula y se acepta la hipótesis alternativa, concluyendo que existe suficiente evidencia para aseverar de que las variables se correlacionan, con un 95% de confianza.

Para la estimación del tiempo hasta la falla funcional (TTF, del inglés *Time To Failure*) del impulsor de la bomba, correspondiente al tiempo en el cual el caudal disminuye hasta 150 l/s, se despeja el tiempo t de la Ecuación (37), obteniendo la siguiente expresión:

$$t = \frac{1}{b_1} ln\left(\frac{Q}{b_0}\right). \quad (40)$$

En la que remplazando los parámetros encontrados y el valor de $Q = 150\ l/s$, se obtiene:

$$t = -\frac{1}{0{,}6} ln\left(\frac{150\ l/s}{342\ l/s}\right)$$

Código:

```
TTF = 1/b1*np.log(150/b0)
año = int(TTF)
mes = int(TTF%1*12)
dia = int((TTF%1*12)%1*30)
print('El impulsor falla en %.0f año, %.0f meses y %.0f dias' % (año, mes, dia))
```

Salida:

El impulsor falla en 1 año, 4 meses y 3 días

Hay que recordar que este resultado es una aproximación por lo que, una vez instalada la bomba, se debe implementar una tarea preventiva donde periódicamente se registre el caudal y posteriormente hacer este análisis de tendencias y monitoreo estadístico del estado del impulsor.

8.3 Conclusiones

El análisis descriptivo de los datos permitió observar que existían valores nulos y atípicos en las muestras de los 4 caudales. De no haberse hecho el tratamiento respectivo no se hubieran ejecutado las funciones empleadas en los siguientes pasos.

Los p valor de la prueba de Shapiro Wilks menores a 0,05, proporcionaron evidencia para aseverar que ninguna de las muestras se distribuyeron normalmente con una confianza del 95%; sin embargo, este hecho no afecta al proceso de linealización, puesto que para este fin se trabajó únicamente con las medias de cada muestra.

Los datos se ajustaron satisfactoriamente a la función exponencial ya que sus valores linealizados tuvieron una correlación r de Pearson con un p valor de 0,0027 que es menor a 0,05, por lo que se puede aseverar de que las variables se correlacionan, con una confianza del 95%.

Se pudo estimar el tiempo hasta la falla funcional del impulsor de la bomba en 1 año con 4 meses y 3 días; sin embargo, esta aproximación podría no ser exacta, por lo que se debe monitorear el caudal de la bomba de manera permanente.

Ejercicios propuestos

1. Haga 5 ejemplos relativos al mantenimiento y 5 relativos a la cotidianidad de datos cuantitativos discretos en listas mediante *Python*.

2. Haga 5 ejemplos relativos al mantenimiento y 5 relativos a la cotidianidad de datos cuantitativos continuos en listas mediante *Python*.

3. Haga 5 ejemplos relativos al mantenimiento y 5 relativos a la cotidianidad de datos cualitativos nominales en listas mediante *Python*.

4. Haga 5 ejemplos relativos al mantenimiento y 5 relativos a la cotidianidad de datos cualitativos ordinales en listas mediante *Python*.

5. En un lote de tamaño desconocido de remaches tipo POP diámetro 1/8", se han mezclado dos diferentes longitudes 5/16" y 3/8", en una muestra inicial de 30 remaches se

encontraron 6 de longitud 3/8". Calcular el tamaño de la muestra para estimar la proporción con un error del 4% y una confianza del 95%, con una función en *Python*.

R.: 385.

6. Calcular el tamaño de la muestra para estimar la proporción en el ejercicio anterior, a) si el tamaño de la población es de 400, b) si el tamaño de la población es de 800. Resuelva con una función en *Python*.

   R.: a) 197, b) 260.

7. Una distribuidora ha realizado mediciones de vibraciones en el plano radial de la chumacera del lado rodete de sus ventiladores radiales accionados por un motor asincrónico de 10 hp con transmisión de bandas y poleas, encontrando una desviación estándar poblacional de $\sigma = 0{,}82\ mm/s$. Calcular el tamaño de la muestra para estimar la media con un error de $0{,}25\ mm/s$ y una confianza del 95%, con una función en *Python*.

   R.: 42.

8. Calcular el tamaño de la muestra para estimar la media en el ejercicio anterior, a) si el tamaño de la población es de 100, b) si el tamaño de la población es de 200. Resuelva con una función en *Python*.

   R.: a) 30, b) 35.

9. En un ventilador radial accionado por un motor asincrónico de 10 hp con transmisión de bandas y poleas, que está operando en una fábrica manufacturera, arroja como resultado de 20 mediciones de vibraciones en el plano radial de la chumacera del lado rodete una desviación estándar muestral de $s = 0{,}82\ mm/s$. Calcular el tamaño de la muestra para estimar la media con un error de $0{,}25\ mm/s$ y una confianza del 95%, con una función en *Python*.

   R.: 48.

10. Calcular el tamaño de la muestra para estimar la media en el ejercicio anterior, de la siguiente muestra de vibraciones en mm/s:

```
vibraciones = [5.3, 3.5, 4.9, 3.6, 4.2, 4.5, 3.9, 4.2, 4.9, 2.9, 3.1, 4.7, 4.5, 2.9, 3.7, 2.8, 5.2, 4.4, 2.7, 4.1]
```

    Resuelva con una función en *Python*.

    R.: 48.

11. Copie el siguiente contenido y guarde en un archivo *txt* para luego importar en *Python* y mostrar la información y los 3 primeros registros.

n vibraciones
1 3,8
2 4,6
3 3,2
4 5,1
5 3,9
6 4,2
7 4,6
8 3,5
9 2,3
10 3,9

R.:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   n            10 non-null     int64
 1   vibraciones  10 non-null     float64
dtypes: float64(1), int64(1)
memory usage: 292.0 bytes
None
   n  vibraciones
0  1          3.8
1  2          4.6
2  3          3.2
```

12. Cree una función en *Python* para hacer la tabla de frecuencias de la siguiente muestra:

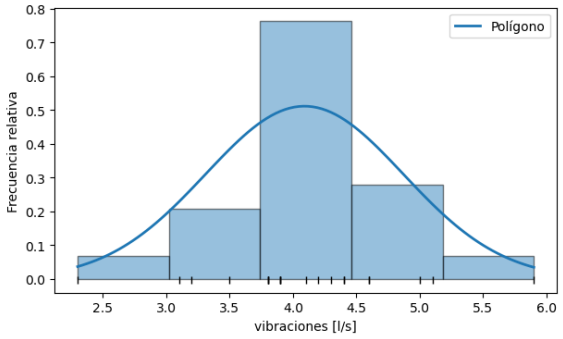
```
vibraciones = [3.8, 4.6, 3.2, 5.1, 3.9, 4.2, 4.6, 3.5, 2.3, 3.9, 4.1, 3.8, 3.8, 4.4, 5.9, 3.9, 3.1, 5.0, 4.4, 4.3]
```

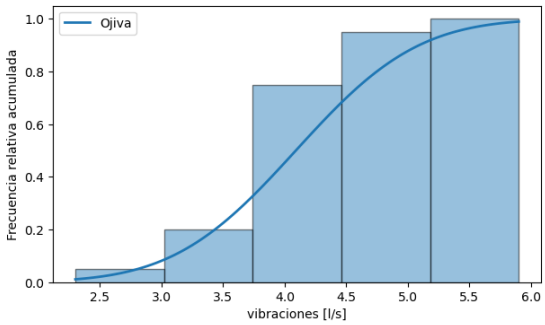
R.:

	Clase	Linf	Lsup	x	f	Fa	fr	Fra
0	1	2.3	3.0	2.65	1.0	1.0	0.05	0.05
1	2	3.0	3.7	3.35	3.0	4.0	0.15	0.20
2	3	3.7	4.4	4.05	11.0	15.0	0.55	0.75
3	4	4.5	5.2	4.85	4.0	19.0	0.20	0.95
4	5	5.2	5.9	5.55	1.0	20.0	0.05	1.00

13. Elabore el histograma de frecuencias relativas y el histograma de frecuencias acumuladas con los datos del ejercicio 12.

R.:





14. Elabore una función en *Python* para calcular con 2 decimales, las medidas de tendencia central: media aritmética, media abreviada de $A = 10$, media geométrica, media armónica, media cuadrática, moda y mediana, y aplíquela en la muestra de vibraciones del ejercicio 12.

    R.:

    Media aritmética:4.09

    Media abreviada :4.09

    Media geométrica:4.02

    Media armónica :3.93

    Media cuadrática:4.16

    Moda: 3.85

    Mediana:4.00

15. Elabore una función en *Python* para calcular con 2 decimales, las medidas de dispersión y posición: rango, desviación estándar, varianza, coeficiente de variación, cuartil 1, cuartil 2 y cuartil 3, y aplíquela en la muestra de vibraciones del

ejercicio 12.

R.:

Rango: 3.60

Varianza: 0.61

Desviación estándar: 0.78

Coeficiente de variación: 19.09

Cuartil 1: 3.80

Cuartil 2: 4.00

Cuartil 3: 4.45

16. Haga una función para identificar los valores atípicos con el método Z-score con un $umbral = 1{,}64$, de la muestra de vibraciones del ejercicio 12.

    R.:

    Valores atípicos: [2.3, 5.9]

17. Haga una función para identificar los valores atípicos con el método del rango intercuartílico de la muestra de vibraciones del ejercicio 12.

    R.:

    Valores atípicos: [2.3, 5.9]

18. Haga una función para imputar los valores atípicos con el

reemplazo de la mediana a la muestra de vibraciones del ejercicio 12.

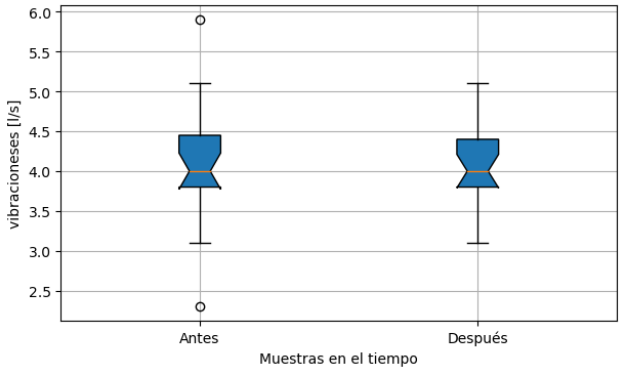
R.:

Mediana sin atípico: 4.0

Datos imputados: [3.8, 4.6, 3.2, 5.1, 3.9, 4.2, 4.6, 3.5, 4.0, 3.9, 4.1, 3.8, 3.8, 4.4, 4.0, 3.9, 3.1, 5.0, 4.4, 4.3]

19. Elabore un solo gráfico con los diagramas de caja de la muestra del ejercicio anterior con el antes y el después de la imputación de los valores atípicos.

R.:



20. Determine y analice el coeficiente de simetría de la muestra de vibraciones del ejercicio 12.

R.:

Coeficiente de simetría: 0.04172
Como el resultado es positivo, la distribución está ...

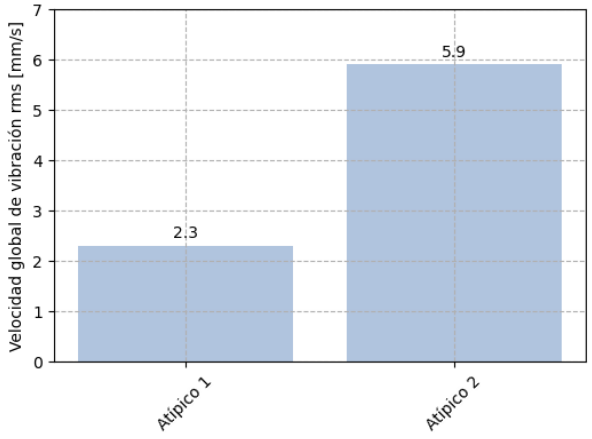
21. Determine y analice la curtosis de la muestra de vibraciones del ejercicio 12.

    R.:

    Curtosis: 1.24957
    Como el resultado es menor que 3, la distribución está ...

22. Haga una función que elabore un diagrama de barras de color *lightsteelblue*, para comparar los valores atípicos de la muestra de vibraciones del ejercicio 12.

    R.:



Conclusiones

- Las indicaciones claras sobre la instalación de los software, extensiones y librerías son claves para el buen desempeño de estos; ya que, por no estar todo integrado en un solo instalador, se debe seguir con cuidado el orden correcto.

- Las librerías de *Python* empleadas en la estadística son pocas; sin embargo, contienen una nutrida variedad de funciones, poniendo a la disposición de los usuarios una gama importante de posibilidades. No todas se tratan en este libro, por lo que es de mucha ayuda la sintetización del uso y configuración de estas funciones y sus parámetros, que se tabulan en el capítulo 1 .

- En el campo del mantenimiento industrial se trabaja principalmente con datos cuantitativos pertenecientes a poblaciones infinitas, por lo que se debe preferencialmente calcular el tamaño de la muestra empleando la distribución T Student ya que aunque se cuente con muchos datos, *Python* no tiene restricciones al trabajar con este estadístico.

- La importación de los datos a *Python* es muy sencilla, para lo que se emplea la librería *Pandas*, que mediante la función *DataFrame,* agrupa a los datos en un arreglo configurado igual que una tabla.

- Las librerías *NumPy* y *Pandas* poseen funciones directas para obtener las medidas de tendencia central, dispersión y posición, por lo que se simplifica muy considerablemente la obtención de estos indicadores; sin embargo, en esta obra se presentan alternativas para realizar los cálculos utilizando las ecuaciones estadísticas con la finalidad de que el lector genere habilidades con el uso del código en *Python*.

- Los métodos gráficos para el análisis de la forma de la distribución no son tan concluyentes como los métodos analíticos; sin embargo, permiten obtener una breve idea de este análisis, por lo que deben realizarse de manera complementaria de los métodos analíticos.

- La imputación de datos faltantes mediante los métodos de eliminación de registros y de imputación simple, pueden realizarse fácilmente utilizando funciones directas de la librería

Pandas como *dropna*() para la eliminación de registros o *fillna*() para la imputación simple.

- Para la imputación de los datos faltantes mediante el método de los k-vecinos se puede utilizar la función *KNNImputer*() de la librería *Sklearn*.

- La identificación de los valores atípicos debe realizarse previo el análisis de que sean reconocidos como tales; ya que, en algunos casos, la presencia de estos son reflejos de la realidad del experimento y no corresponden a errores de medida.

- Las técnicas más comunes para identificar valores atípicos son el método del rango intercuartílico y el método Z-Score que pueden implementarse utilizando funciones de recurrencia.

- Para optar entre los métodos de eliminación e imputación para el tratamiento de los valores atípicos, se debe ir comprobando la distribución de los datos resultantes para decidir el que mejor se ajuste.

- Para visualizar la distribución y tendencia de los datos, así

como sus principales indicadores o para realizar comparaciones de resultados, existen varias alternativas en las que las librerías *Matplotlib* y *Seaborn* son las más utilizadas.

- El caso práctico permitió proveer alternativas de funciones directas para la exploración, análisis de la distribución, tendencia y regresión lineal de los datos para la solución de un problema real de mantenimiento basado en la condición de un sistema de bombeo.

Glosario

A

***a*:** ancho de clase.

Algoritmo: secuencia de pasos que da solución a un problema.

Asimetría (skewness): es un coeficiente que mide la existencia de un sesgo en la distribución de los datos de tal manera que si vale 0, la distribución es simétrica.

B

***β*:** es el parámetro de forma de la distribución de Weibull.

Bomba centrífuga: es una máquina que transforma energía mecánica en energía cinética de presión a un fluido.

C

Campana de Gauss: gráfico de la ecuación formulada por Gauss que tiene forma de campana invertida.

Caudal: cantidad de volumen de fluido por unidad de tiempo.

Código: conjunto de instrucciones propias de un programa para indicar a la máquina secuencias de ejecución.

Curtosis: es una medida del pico y dispersión de una distribución.

CV: coeficiente de variación.

D: decil, del inglés Decile.

D

DataFrame: para manejo de datos, están compuestos por filas y columnas.

Datos faltantes: son cuando en una muestra faltan valores.

Disponibilidad: aptitud de un elemento para permanecer en un estado en que pueda cumplir con su función requerida.

E

e: es el número de Euler que es de 2,7182818284…

E(x): esperanza matemática o media.

***η*:** es el parámetro de escala de la distribución de Weibull.

ERP: software de gestión empresarial, del inglés *Enterprise Resource Planning*.

F

f: frecuencia absoluta.

Fa: frecuencia absoluta acumulada.

fr: frecuencia relativa.

Fra: frecuencia relativa acumulada.

Función inherente: es lo que un activo físico es capaz de hacer por sus características de diseño en un contexto operacional determinado.

F(t): es la probabilidad acumulada de ocurrencia de un evento.

G

G: media geométrica.

GMAO: gestión del mantenimiento asistido por computador.

H

H: media armónica.

L

Lenguaje de programación: programa con reglas y gramáticas propias del programa, para generar nuevos programas.

Leptokurtic: es cuando la curtosis de una distribución es mayor que 3, siendo más dispersa que la distribución normal.

Longitud: extensión en línea recta.

M

Mantenimiento predeterminado: es la sustitución o restauración de un elemento a intervalos fijos sin la evaluación de su estado.

Markdown: es un lenguaje de marcado que facilita la aplicación de formato a un texto.

Me: mediana.

Mesocúrtica: es cuando la curtosis de una distribución es igual que 3, siendo igual de dispersa que la distribución normal.

***μ*:** media poblacional.

Mo: moda.

N

n: tamaño de la muestra.

N: tamaño de la población.

P

P: percentil, del inglés *Percentile*.

Platykurtic: es cuando la curtosis de una distribución es menor que 3, siendo menos dispersa que la distribución normal.

Presión: fuerza ejercida sobre un área determinada.

***Python*:** lenguaje de programación.

Q

q: cuantil, del inglés *quantile*.

Q: cuartil, del inglés Quartile.

R

RCM: Raíz Cuadrada Media.

RIC: rango intercuartílico.

RMS: Raíz media cuadrada, del inglés *Root Mean Square*.

Ruido: perturbación aperiódica o de frecuencia variable.

S

***s*:** desviación estándar muestral.

***s*^{2}:** varianza muestral.

***σ*:** desviación estándar poblacional.

***σ*^{2}:** desviación estándar poblacional.

String: cadena de caracteres.

T

***TBF*:** tiempo entre fallas, del inglés *Time Between Failure*.

Temperatura: medida de la energía interna de la materia.

***TTF*:** tiempo hasta la falla funcional, del inglés *Time To Failure*.

***TTR*:** tiempo hasta la recuperación, del inglés *Time To Restoration*.

V

Valores atípicos (outliers): son los valores que se alejan demasiado de las medidas de tendencia central.

***VAR(x)*:** varianza.

Variable: es un elemento que puede tomar cualquier valor de los comprendidos en un conjunto.

Velocidad: tasa de cambio de la posición en función del tiempo.

Vibración: movimiento oscilatorio periódico.

X

***x*:** marca de clase.

***$\bar{x}$*:** media muestral.

Referencias

Amat, J. (2021). *Análisis de normalidad con Python*. https://cienciadedatos.net/documentos/pystats06-analisis-normalidad-python

Charlie, E. (2023). *Openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files*. https://openpyxl.readthedocs.io/en/stable/

Coder, R. (2023). *Python Charts*. https://python-charts.com/es/

Coursera. (2023). *¿Qué es Python y para qué se usa? Guía para principiantes*. https://www.coursera.org/mx/articles/what-is-python-used-for-a-beginners-guide-to-using-python

Johnson, R. (2012). *Probabilidad y Estadística para Ingenieros* (8va ed). Pearson Educación de México, S.A.

Kelmansky, D. (2009). *Estadística para todos. Estrategias de pensamiento y herramientas para la solución de problemas* (1ra ed). Artes gráficas Rioplatense S. A.

Lifeder. (2020). *Población estadística: concepto, tipos, ejemplos*. https://www.lifeder.com/poblacion-estadistica/

Lind, D., Marchal, W., & Wathen, S. (2012). *Estadística aplicada a los negocios y la economía* (15ta ed). McGraw-Hill Companies, Inc.

Matplotlib. (2023). *Matplotlib: Visualization with Python*. https://matplotlib.org/

Maximov, L. (2020). *NumPy Illustrated: The Visual Guide to NumPy*. https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d

Mendenhall, W., Beaver, R., & Beaver, B. (2015). *Introducción a la Probabilidad y Estadística* (14a ed). Cengage Learning Editores.

Nube de datos. (2015). *Introducción al diagrama de caja (box plot) en R*. https://nubededatos.blogspot.com/2015/02/introduccion-al-diagrama-de-caja-box.html

Pandas. (2023). *Pandas - Getting started*. https://pandas.pydata.org/docs/getting_started/index.html

Pértegas, S., & Pita, S. (2001). *Guía: La distribución normal - Fisterra*. https://www.fisterra.com/formacion/metodologia-investigacion/la-distribucion-normal/

Prasad, D. (2023). *Los 10 mejores IDE de Python para potenciar el desarrollo y la depuración*. https://geekflare.com/es/best-python-ide/

Python. (2023). *Descargar la última versión para Windows*. https://www.python.org/downloads/

Rodríguez, D. (2021). *Imputación de valores nulos en Python*. https://www.analyticslane.com/2021/03/22/imputacion-de-valores-nulos-enpython/

Rootstack. (2022). *Los mejores entornos de desarrollo de Python para utilizar en el 2023*. https://rootstack.com/es/blog/los-mejores-entornos-de-desarrollo-de-python-para-utilizar-en-el-2023

Rougier, N. (2017). *From Python to Numpy*. https://doi.org/10.5281/zenodo.225783

Scikit-Learn. (2023). *Scikit-Learn. Machine Learning in Python*. https://scikit-learn.org/stable/

SciPy. (2023a). *SciPy User Guide*. https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide

SciPy. (2023b). *Statistics (scipy.stats)*. https://docs.scipy.org/doc/scipy/tutorial/stats.html

Seaborn. (2022). *Seaborn: statistical data visualization*. https://seaborn.pydata.org/

Spiegel, M., & Stephens, L. (2009). *Estadística - Schaum* (4ta ed.). McGraw-Hill Companies, Inc.

Statologos. (2021). *Cómo calcular la Asimetría y la Curtosis en Python*. https://statologos.com/sesgo-curtosis-piton/

Triola, M. (2018). *Estadística* (12da ed). Pearson Educación.

UNE-EN 13306. (2018). *Mantenimiento - Terminología del mantenimiento*. AENOR INTERNACIONAL S.A.U. www.aenor.com

Valverde, V. L., Cajamarca, J. E., & Moreano, G. V. (2023). *Fundamentos de Programación con DFD-PSeInt-Python*. Editorial CIDE. https://doi.org/https://doi.org/10.33996/cide.ecuador.PS2636324

Valverde, V. L., García, F. A., & Hernández, E. S. (2023). *Python aplicado al mantenimiento industrial*. Editorial CIDE. https://doi.org/https://doi.org/10.33996/cide.ecuador.PA2636331

Visual Studio Code. (2023a). *Download Visual Studio Code. Free and built on open source. Integrated Git, debugging and extensions.* https://code.visualstudio.com/download

Visual Studio Code. (2023b). *Python in Visual Studio Code*. https://code.visualstudio.com/docs/languages/python

Estadística descriptiva para el mantenimiento industrial con Python. El objetivo de este libro es ayudar al lector en el desarrollo de habilidades para que pueda aplicar la estadística descriptiva en el mantenimiento industrial mediante el empleo de Python utilizando datos extraídos del comportamiento de máquinas industriales con el propósito de incentivar el pensamiento analítico y crítico en la solución de problemas de la ingeniería del mantenimiento. El capítulo 1 sintetiza la instalación y empleo de los software y librerías requeridas, para luego en el capítulo 2 tratar sobre la obtención y organización de datos; seguidamente en los capítulos 3 y 4 se aborda el cálculo de las medidas de tendencia central, dispersión y posición respectivamente; a continuación, en el capítulo 5 se expone el análisis de la forma de la distribución; seguido del capítulo 6 con el manejo de datos faltantes y atípicos; para posterior a eso, en el capítulo 7 explicar los métodos para la elaboración de gráficos estadísticos; y finalmente en el capítulo 8, desarrollar un caso práctico sobre el mantenimiento basado en la condición de bombas centrífugas.

Eduardo Segundo Hernández Dávila. –Ingeniero de Mantenimiento y Magister en Gestión del Mantenimiento Industrial por la Escuela Superior Politécnica de Chimborazo, en donde actualmente, se desempeña notablemente como docente desde hace 16 años en el área de la ingeniería de mantenimiento y la fiabilidad. Forma parte del Grupo de Investigación Ciencia del Mantenimiento "CIMANT", aportando en la investigación y transferencia de tecnologías en el área operativa y de gestión del mantenimiento industrial. En empresas manufactureras ha ejercido cargos en la dirección del mantenimiento por 5 años.

César Marcelo Gallegos Londoño. – Ingeniero de Mantenimiento Industrial, por la Escuela Superior Politécnica de Chimborazo; Maestría en Gestión del Mantenimiento Industrial. Docente de la Escuela Superior Politécnica de Chimborazo. Experiencia en montajes eléctricos 9 años; implementación de Software de mantenimiento por 17 años. Forma parte del grupo de Investigación "GIOMANT".

Félix Antonio García Mora. – Ingeniero de Mantenimiento con una destacada carrera en la ingeniería y la gestión del mantenimiento industrial. Obtuvo su título en la Escuela Superior Politécnica de Chimborazo y una Maestría en Gerencia e Ingeniería de Mantenimiento en la Universidad Nacional de Ingeniería en Lima, Perú. Ha ocupado cargos importantes en la dirección del mantenimiento en diversas industrias, además de dedicarse a la enseñanza durante 3 años. Es miembro del Grupo de Investigación Ciencia del Mantenimiento (CIMANT), lo que refleja su interés en la investigación y el avance de las mejores prácticas en el campo de la ingeniería y la gestión de mantenimiento.

ISBN: 978-9942-636-55-3
9789942636553