Representaciones gráficas de datos

En este documento veremos algunas librerías de Python que nos van a permitir obtener distintas representaciones gráficas del conjunto de datos con el que estamos trabajando. Nos centraremos sobre todo en *Matplotlib*, que es la librería más popular en este sentido, pero también veremos pinceladas de alguna otra, como Seaborn.

Cuando trabajamos con Machine Learning, es esencial comprender los datos, evaluar modelos, visualizar resultados y muchas más cosas en las que **Matplotlib** supone una herramienta imprescindible, como, por ejemplo:

- Crear gráficos de dispersión para identificar patrones o correlaciones.
- Visualizar distribuciones con histogramas.
- Detectar valores atípicos (outliers) mediante diagramas de caja (**boxplots**).
- Gráficos de predicción vs. valor real para regresiones.
- Visualizar clústeres.

Contenido

1. *Matplotlib*

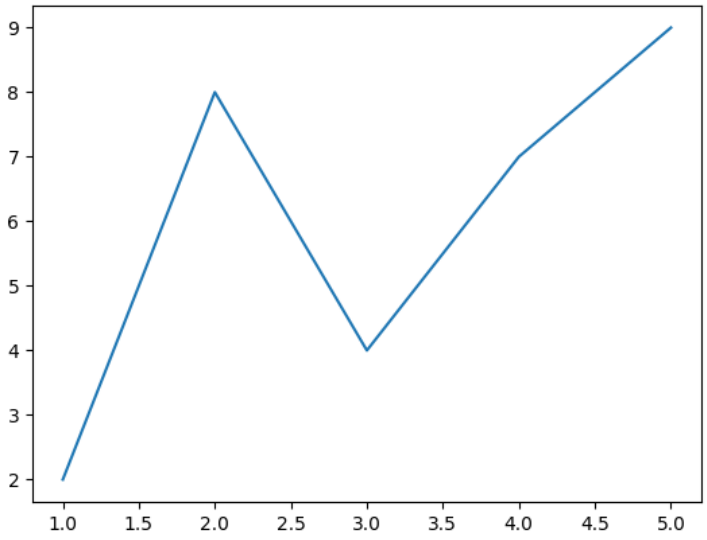
Matplotlib es una librería externa al núcleo de Python que permite obtener distintos tipos de gráficos para un conjunto de datos. Es probable que tengamos que instalarla. Ofrece una gran cantidad de opciones, cosa que podemos comprobar consultando su documentación oficial. Aquí nos centraremos en unos pocos ejemplos básicos, para lo que nos servirá con el componente `pyplot` de la librería.

1.1. Tipos de gráficos básicos

Veamos a continuación algunos de los gráficos más habituales que podemos representar con la librería.

1.1.1. Gráfico de líneas

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 8, 4, 7, 9]
plt.plot(x, y)
plt.show()
```

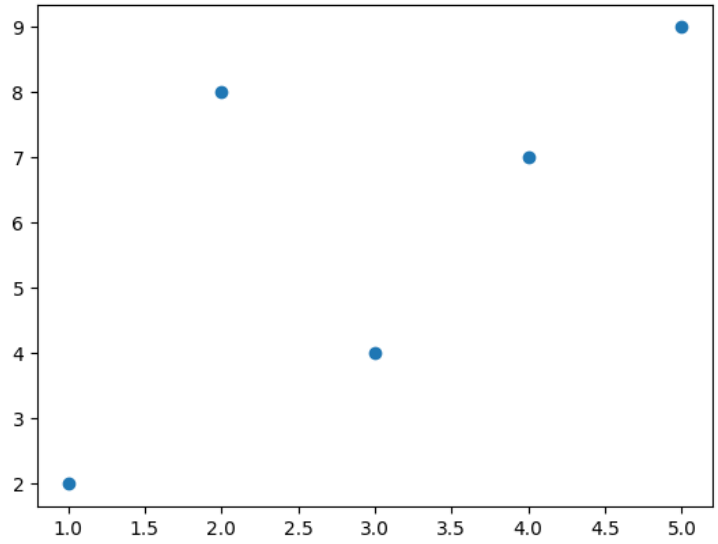


1.1.2. Gráficos de dispersión (secuencias de puntos)

Son habituales en operaciones de regresión lineal donde, en base a pares de valores previos, se debe inferir qué valor corresponde a uno nuevo dado. También se utilizan para determinar si existe una correlación entre un par de parámetros.

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 8, 4, 7, 9]
```

```
# Saca los puntos (1, 2), (2, 8), (3, 4)... etc
plt.scatter(x, y)
plt.show()
```

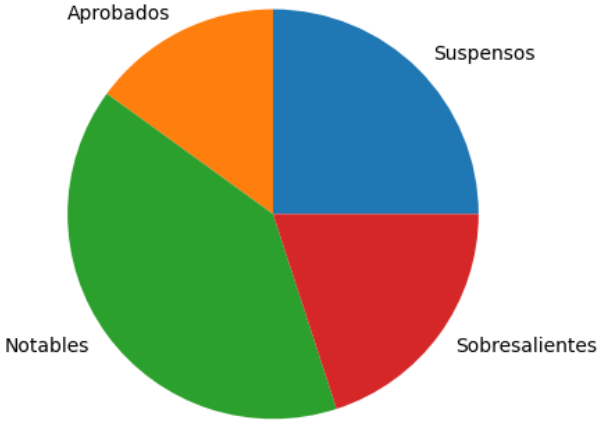


Se pueden configurar algunas cosas adicionales, como establecer diferentes colores para cada punto, o incluso dibujar los puntos con un tamaño u otro según los valores numéricos asociados.

1.1.3. Gráficos circulares o de sectores

Los gráficos circulares tienen una particularidad: se construyen a partir de un conjunto de valores numéricos dados, representando entonces el porcentaje que supone cada valor con respecto al total de ellos.

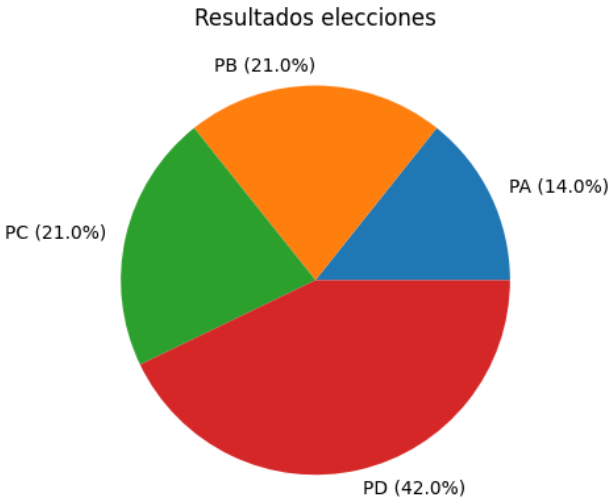
```
import matplotlib.pyplot as plt

calificaciones = ['Suspensos', 'Aprobados', 'Notables', 'Sobresalientes']
valores = [5, 3, 8, 4]

plt.pie(valores, labels=calificaciones)
plt.show()
```



Podemos configurar algunas cosas adicionales de este tipo de gráficos. Por ejemplo, el parámetro `explode` nos puede servir para destacar algunas porciones sobre otras ("sacarlas" de la tarta), el parámetro `colors` permite especificar los colores de cada porción y el parámetro `autopct` permite mostrar el valor porcentual.

Ejercicio 1

Crea un programa **Elecciones.py** que va a representar en un gráfico los resultados de unas elecciones a las que concurren cuatro partidos políticos, PA, PB, PC y PD. El usuario deberá introducir en una línea los votos recibidos, en forma de números enteros que representan a cada partido (1 = PA, 2 = PB, 3 = PC, 4 = PD, por ejemplo: `1 1 2 1 1 2 4 3 2 1`), y mostrará un gráfico circular con el resultado final de las elecciones. Junto al nombre de cada partido debe aparecer el porcentaje de voto total. Cualquier número que no esté en el rango indicado (de 1 a 4) deberá descartarse del conteo final. La imagen del ejemplo no corresponde a los valores numéricos anteriores.



1.1.4. Histogramas

Los histogramas son gráficos de barras que representan la frecuencia de cada dato.

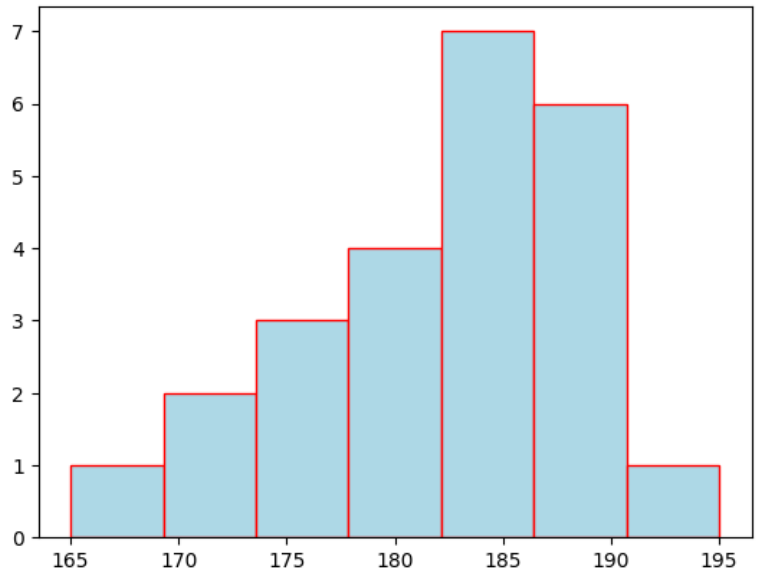
```
import matplotlib.pyplot as plt

# Histograma de frecuencias de alturas en un grupo de personas
alturas = [185, 170, 185, 190, 175, 190, 185, 175,
    180, 180, 185, 190, 175, 190, 185, 170, 165, 195,
    180, 185, 190, 190, 185, 180]
plt.hist(alturas)
plt.show()
```

En el caso de representar valores numéricos en el eje horizontal podemos indicar la cantidad de intervalos, número de cajas o barras en que queremos dividir el conjunto de valores, a través del parámetro `bins`:

```
# Mostraremos 7 barras agrupando los valores
plt.hist(alturas, bins=7)
plt.plot()
```

Podemos también especificar un color para los bordes y otro para las barras (y así diferenciar mejor cada barra) con los parámetros `edgecolor` y `color`, respectivamente:

```
# Mostraremos 7 barras agrupando los valores
plt.hist(alturas, bins=7, color='lightblue', edgecolor='red')
plt.plot()
```

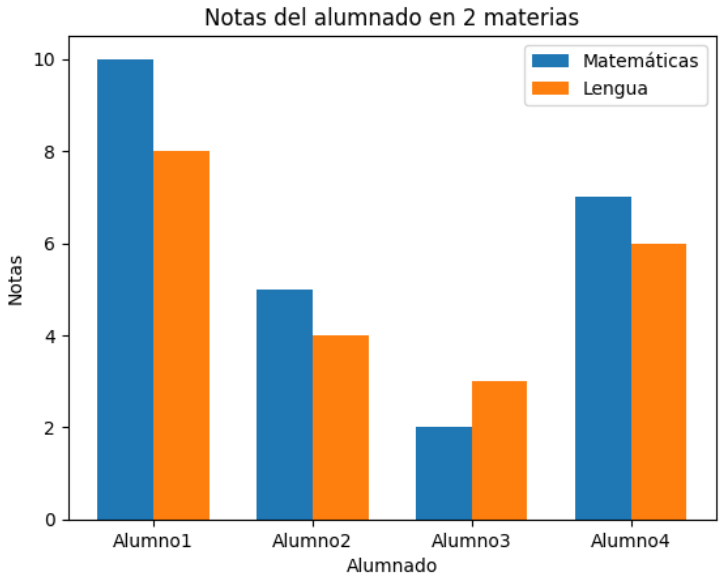


1.1.5. Gráficos de barras

Los gráficos de barras representan la cantidad de elementos de diferentes categorías, y las barras aparecen separadas unas de otras (a diferencia de los histogramas). Aquí vemos otro ejemplo con un gráfico de barras donde representamos dos secuencias de datos. Usamos para ello la función `bar` para cada secuencia de barras:

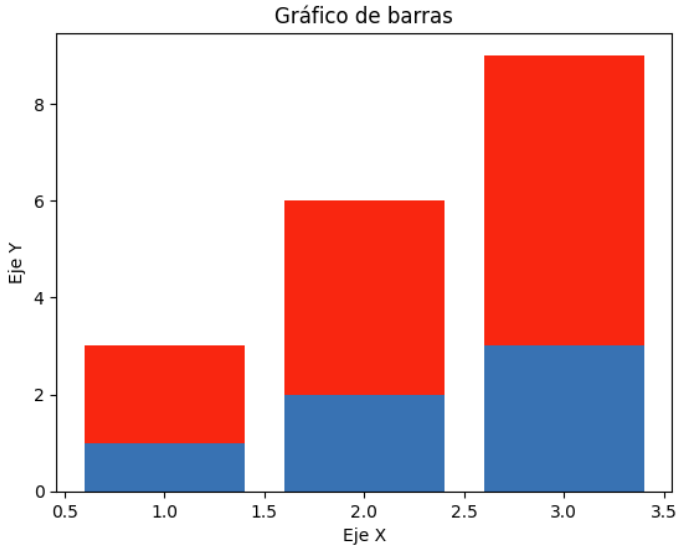
```
categories = ['Alumno1', 'Alumno2', 'Alumno3', 'Alumno4']
values1 = [10, 5, 2, 7]
values2 = [8, 4, 3, 6]

bar_width = 0.35
x = np.arange(len(categories))

plt.bar(x - bar_width/2, values1, bar_width, label='Matemáticas')
plt.bar(x + bar_width/2, values2, bar_width, label='Lengua’)

plt.xlabel('Alumnado')
plt.ylabel('Notas')
plt.title('Notas del alumnado en 2 materias')
plt.xticks(x, categories)
```

```
plt.legend()
plt.show()
```



Alternativamente podemos mostrar las barras acumuladas una sobre otra (si comparten los mismos valores de X), usando el parámetro `bottom` para indicar el punto de inicio de cada barra respecto a la anterior:

```
x = [1,2,3]
y = [1,2,3]
x2 = [1,2,3]
y2 = [2,4,6]

plt.bar(x, y)
plt.bar(x2, y2, bottom = y, color = 'r')
plt.title('Gráfico de barras')
plt.ylabel('Eje Y')
plt.xlabel('Eje X')
plt.show()
```



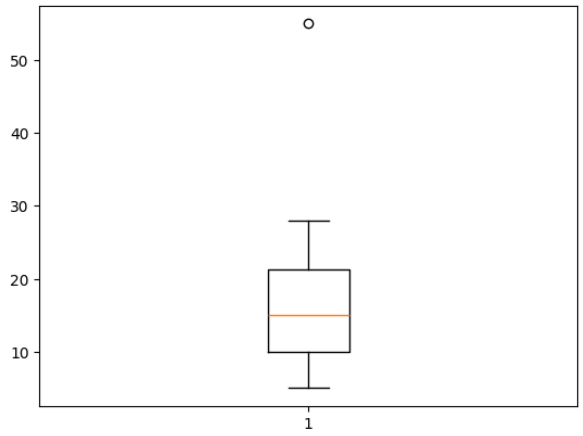
Para mostrar el gráfico **horizontal** usaremos la instrucción `barh` en lugar de `bar`. En este caso, si queremos mostrar barras acumuladas hay que tener en cuenta que se acumulan desde la izquierda (`left`) en lugar de desde abajo (*bottom*):

```
x = [1,2,3]
y = [1,2,3]
x2 = [1,2,3]
y2 = [2,4,6]

plt.barh(x, y)
plt.barh(x2, y2, left = y, color = 'r')
plt.title('Gráfico de barras')
plt.ylabel('Eje Y')
plt.xlabel('Eje X')
plt.show()
```

1.1.6. Gráficos de cajas

Los gráficos de cajas (*box plots*) permiten ver la distribución de valores de una determinada categoría (o varias) para determinar si están muy juntos o dispersos, y también si existen valores anómalos por encima o por debajo de lo normal. Para representarlos necesitamos una secuencia de valores numéricos, y llamar con ella a la instrucción `boxplot`:

```
x = [10, 20, 15, 12, 15, 20, 25, 10, 55, 20, 10, 8, 28, 13, 26, 5]
plt.boxplot(x)
plt.show()
```



Podemos acumular varias secuencias de valores, y asignarle a cada una un rótulo o etiqueta para el eje X:

```
x1 = [10, 20, 15, 12, 15, 20, 25, 10, 90, 20, 10, 8, 28, 13, 26, 5]
x2 = [40, 41, 43, 44, 46, 48, 5, 38, 50, 40, 39, 49]
plt.boxplot([x1, x2], labels=["x1", "x2"])
plt.show()
```

Las cajas centrales de cada categoría se denominan *rango intercuartil* (IQR), y acumulan el 50% central de los valores de la muestra. También indican el grado de dispersión de esos valores (cuanto más "chata" es la caja, más agrupados

están). Los bigotes superior e inferior indican la extensión de los datos más allá de ese núcleo central. Finalmente, los puntos aislados por encima o debajo de los bigotes representan valores anómalos, demasiado alejados de la concentración principal de valores.

1.2. Opciones de configuración

En algún ejemplo anterior hemos visto que podemos especificar algunas características de los gráficos, tales como colores, textos de los ejes, alineaciones... *Matplotlib* incorpora una serie de parámetros para configurar aspectos como colores, leyendas, ejes, etc. Veamos ahora algunos ejemplos de ello.

1.2.1. Títulos del gráfico y de los ejes

Para especificar el título general del gráfico, tenemos una función `title`:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 8, 4, 7, 9]
plt.plot(x, y)
plt.title('Gráfico de prueba')
plt.show()
```

Adicionalmente, podemos elegir la ubicación del título, con el parámetro `loc`:

```
plt.title('Gráfico de prueba', loc='left')
```

Para rotular los ejes, tenemos las funciones `xlabel` e `ylabel`, respectivamente para cada eje, como hemos visto en un ejemplo anterior:

```
import matplotlib.pyplot as plt

x = [5,8,10]
y = [12,16,6]
x2 = [6,9,11]
y2 = [6,15,7]

plt.bar(x, y, align = 'center')
plt.bar(x2, y2, color = 'g', align = 'center')
plt.title('Gráfico de barras')
plt.ylabel('Eje Y')
plt.xlabel('Eje X')
plt.show()
```

Adicionalmente podemos especificar el tipo de letra para cada eje. Para ello crearemos un diccionario que especifique la familia, color y tamaño de la fuente, y asignaremos ese diccionario a cada título o etiqueta con el parámetro `fontdict`:

```
font1 = {'family': 'Arial', 'color': 'red', 'size': 20}
font2 = {'family': 'Courier', 'color': 'blue', 'size': 14}

plt.title('Gráfico de barras', fontdict = font1)
plt.ylabel('Eje Y', fontdict = font2)
plt.xlabel('Eje X', fontdict = font2)
```

Ejercicio 2

Crea un programa **Ventas.py** que le pregunte al usuario por las ventas de una empresa en un rango de años (indicado por el usuario) y muestre un diagrama de líneas con la evolución.

1.2.2. Colores, estilos de línea y marcadores

Los **colores** de los elementos (líneas, barras, etc) en *Matplotlib* se codifican normalmente con una letra minúscula, aunque también se pueden especificar con un código hexadecimal. Dependiendo del elemento sobre el que aplicar el color, se hace de una forma u otra. Por ejemplo, para especificar el color de una línea en un gráfico lineal, usamos el atributo `color`.

```
# Color rojo
plt.plot(x, y, color='r')
# Color hexadecimal
plt.plot(x, y, color='#428A92')
```

Aquí podéis consultar una serie de colores disponibles directamente en *Matplotlib*.

En cuanto a los **estilos de línea**, podemos especificar un atributo `linestyle` o `ls` con el mismo, que por defecto es sólido (*solid*). También podemos cambiar el grosor de la línea con `linewidth`.

```
plt.plot(x, y, linestyle='dotted', linewidth=5)
```

Aquí podéis consultar un listado con las opciones disponibles.

Los **marcadores** permiten identificar puntos clave en un gráfico. Podemos utilizar el parámetro `marker` para enfatizar esos puntos clave.

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 8, 4, 7, 9]
plt.plot(x, y, marker='o')
plt.show()
```

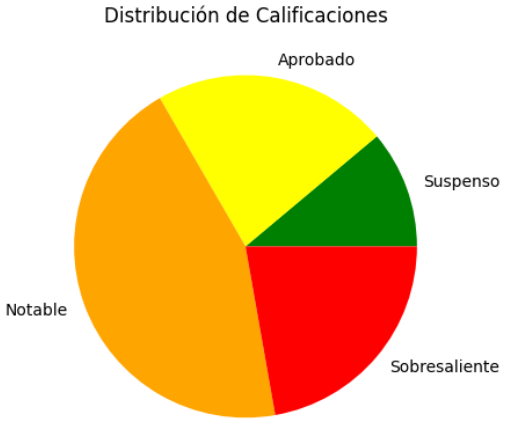
Existen otros símbolos que podemos poner como marcadores, tales como asteriscos `*`, equis `x`, etc. Aquí podemos consultar un listado más exhaustivo.

De forma compacta podemos modificar estas tres opciones a la vez en un único parámetro de texto, indicando color, estilo de línea y marcador:

```
plt.plot(x, y, 'c-o')
```

Ejercicio 3

Crea un programa **GraficoCircularNotas.py** que le pida al usuario una secuencia de notas numéricas (en una línea, separadas por espacios) y muestre un gráfico circular con el porcentaje de suspensos, aprobados, notables y sobresalientes. Personaliza el color de cada porción del gráfico a tu gusto. Por ejemplo:



1.2.3. Leyendas

Existen varias formas de definir la leyenda de un gráfico. Una de las más cómodas quizá consista en definir en cada componente su etiqueta de leyenda con un parámetro `label`, y luego llamar al método `legend` para indicar que queremos mostrar la leyenda con las etiquetas, y en qué ubicación con el parámetro `loc`.

El siguiente ejemplo muestra dos líneas (roja y verde) con sus respectivas etiquetas, y ubica la leyenda en la parte superior central del gráfico:

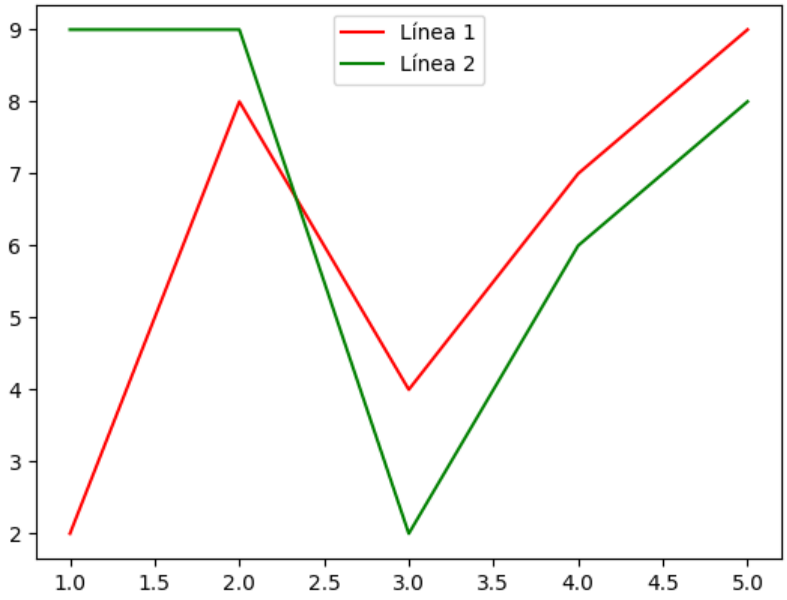
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 8, 4, 7, 9]
x2 = [1, 2, 3, 4, 5]
y2 = [9, 9, 2, 6, 8]
plt.plot(x, y, label='Línea 1', color='r')
plt.plot(x2, y2, label='Línea 2', color='g')
plt.legend(loc='upper center')
plt.show()
```

De forma alternativa, podemos especificar un vector de etiquetas en el método `legend`, en lugar de etiquetar cada gráfico con su `label`. Las etiquetas

del vector se asignan a cada gráfico en el mismo orden en que se van dibujando (el orden de los `plot`):

```
plt.legend(['Línea 1', 'Línea 2'], loc='upper center')
```

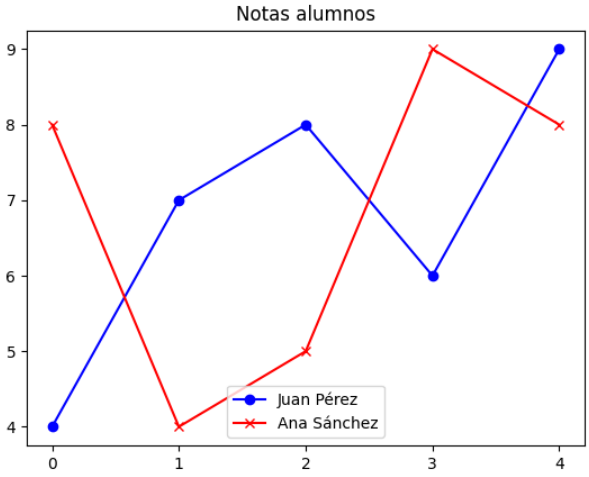
Obtendremos un resultado como este:



Ejemplos de posibles ubicaciones usadas en `loc` son *upper right*, *upper center*, *upper left*, *center left*, *lower right*, etc.

Ejercicio 4

Crea un programa llamado **GraficoNotas.py** que le pida al usuario una secuencia de notas numéricas de dos alumnos (el mismo número de notas para ambos), separadas por espacios, junto con sus nombres (en otra línea aparte). Después muestra en un gráfico de líneas las notas introducidas. Utiliza colores y marcadores diferentes para cada alumno, y una leyenda con sus nombres. Debe quedarte algo así:



1.2.4. Valores de los ejes

Por defecto los ejes de los gráficos muestran unos valores extremos acordes a los mínimos y máximos para ese eje, y una división horizontal/vertical predefinida, dependiendo del conjunto de valores representado. Pero ambas cosas se pueden configurar y personalizar.

Por un lado, las funciones `xlim` e `ylim` nos servirán para delimitar los valores mínimo y máximo de cada eje en un gráfico bidimensional

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 8, 4, 7, 9]
plt.plot(x, y)
plt.xlim(1, 9)
plt.ylim(2, 20)
plt.show()
```

Además, podemos cambiar los valores que se muestran en cada punto del eje X e Y con las funciones `xticks` e `yticks`. Si queremos reemplazar los valores numéricos por defecto por otros, basta con que indiquemos como parámetro el nuevo vector de valores a mostrar. Si queremos hacer un cambio de tipo (convertir, por ejemplo, un eje numérico en uno categórico), deberemos indicar la lista de valores inicial y la lista de valores con que reemplazarlos:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
valores = ['A', 'B', 'C', 'D', 'E']
y = [2, 8, 4, 7, 9]
plt.plot(x, y)
plt.xlim(1, 9)
plt.ylim(2, 20)
plt.xticks(x, valores)
# Mostrará las letras en lugar de los números de 1 a 5
plt.show()
```

Adicionalmente, estas funciones tienen un parámetro `rotation` que permite girar los rótulos:

```
plt.xticks(x, valores, rotation=90)
```

También podemos hacer uso de funciones de *NumPy* para generar la secuencia de valores automáticamente. Por ejemplo, así definimos una secuencia del 1 al 10 con intervalos de 0.5:

```
plt.xticks([x for x in np.arange(1, 10.5, 0.5)])
```

1.2.5. Tamaño de los gráficos y múltiples gráficos

Para modificar el tamaño de un gráfico haremos uso del método `figure`, combinado con otras opciones. En este sentido tenemos dos alternativas:

- Utilizar los métodos `set_figwidth` y `set_figheight` para especificar por separado la anchura y altura del gráfico
- Utilizar un parámetro `figsize` dentro del método `figure` para indicar ambos tamaños en una tupla

El tamaño indicado viene en unidades de medida que por defecto son *pulgadas*, aunque se puede modificar con el parámetro `dpi` para cambiar los puntos por pulgada.

Aquí vemos un par de ejemplos equivalentes:

```
# Establecemos un tamaño de gráfico de 4 unidades de ancho y 2 de alto

# Opción 1
f = plt.figure()
f.set_figwidth(4)
f.set_figheight(2)

# Opción 2
plt.figure(figsize=(4, 2))
```

Matplotlib también permite definir **múltiples gráficos** en un solo dibujado, a través de los `subplots`. Para empezar, tendremos que establecer el número de filas y columnas que queremos rellenar con gráficos, en los parámetros `nrows` y `ncols`. También podemos especificar el tamaño total de la figura que contendrá a todos los gráficos, en el parámetro `figsize`.

```
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize = (10, 6))
```

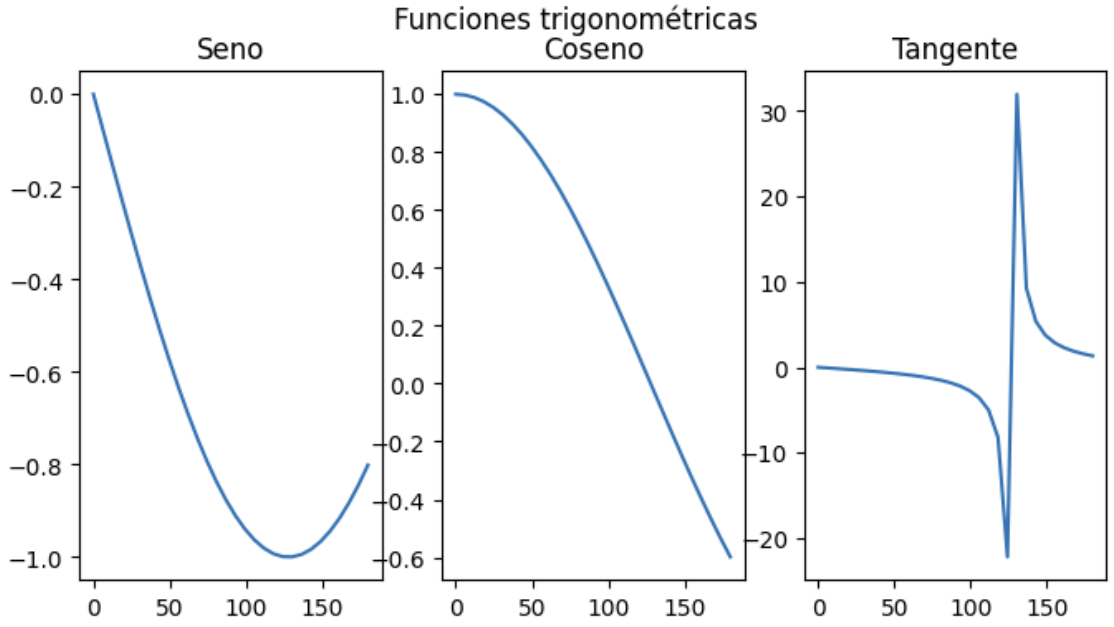
El resultado de la llamada a `subplots` devuelve la figura (*fig*) o lienzo donde se crearán los gráficos y el eje (o conjunto de ejes) donde se dibujará cada gráfico. En el ejemplo anterior se crearán 4 ejes, donde dibujar cada uno de los gráficos de la matriz 2x2 establecida.

A partir de aquí podemos especificar qué gráfico dibujar en cada coordenada de la matriz formada:

```
# Histograma
axes[0,0].hist(...)
# Circular
axes[0,1].pie(...)
# Barras
axes[1,0].bar(...)
# Dispersión
axes[1,1].scatter(...)

plt.show()
```

Aquí vemos un ejemplo concreto:

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

x = np.linspace(0, 180, 30)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(8, 4))
axes[0].plot(x, y1)
axes[0].set_title("Seno")
axes[1].plot(x, y2)
axes[1].set_title("Coseno")
axes[2].plot(x, y3)
axes[2].set_title("Tangente")
fig.suptitle("Funciones trigonométricas")

plt.show()
```

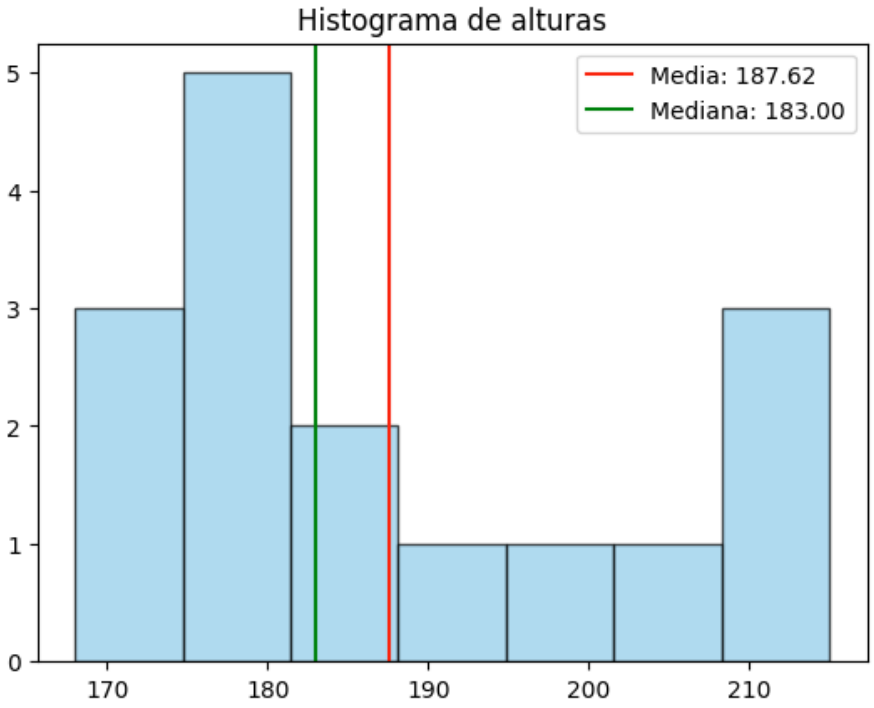


De forma alternativa, también podemos acumular en un solo gráfico varias representaciones gráficas. Por ejemplo, podemos mostrar un histograma con la distribución de valores de un conjunto, y representar con líneas rectas la ubicación de su media y de su mediana. Aquí vemos un ejemplo:

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

# Alturas de un grupo de individuos
datos = np.array([185, 170, 180, 192, 178, 181, 185, 195, 168, \
    172, 210, 215, 210, 181, 202, 178])

# Calculamos la media y la mediana de los datos
media = np.mean(datos)
mediana = np.median(datos)
```

```
# Creamos el histograma
plt.hist(datos, bins=7, alpha=0.7, color='skyblue', edgecolor='black')

# Agregamos una línea vertical en la media y otra con la mediana
plt.axvline(media, color='red', label=f'Media: {media:.2f}')
plt.axvline(mediana, color='green', label=f'Mediana: {mediana:.2f}')

plt.title('Histograma de alturas')
plt.legend()
plt.show()
```

Obtendremos algo como esto:



1.2.6. Otras opciones de configuración

A continuación, explicamos brevemente algunas opciones adicionales de configuración en los gráficos de *Matplotlib*.

En primer lugar, podemos mostrar una **rejilla** (*grid*) que permite ver mejor los valores representados en el gráfico. Esto se consigue con el método `grid`, al que le podemos indicar también el color y tipo de línea de la rejilla a mostrar, de forma similar a como se ha explicado anteriormente.

```
plt.grid(linestyle='dotted', color='k')
```

También podemos cambiar el **estilo** del gráfico, a elegir entre los estilos disponibles en la propiedad `plt.style.available`. Usaremos el método `use` de la propiedad `style`. Por ejemplo así mostraremos el gráfico al estilo de la librería Seaborn:

```
plt.style.use('seaborn')
```

Finalmente, también podemos **guardar la imagen** de la figura en un archivo en nuestro ordenador, con la instrucción `savefig`. Es posible que, según el formato que elijamos (PNG, JPEG...) se guarde más o menos información del gráfico.

```
plt.savefig('figura1.jpeg')
```

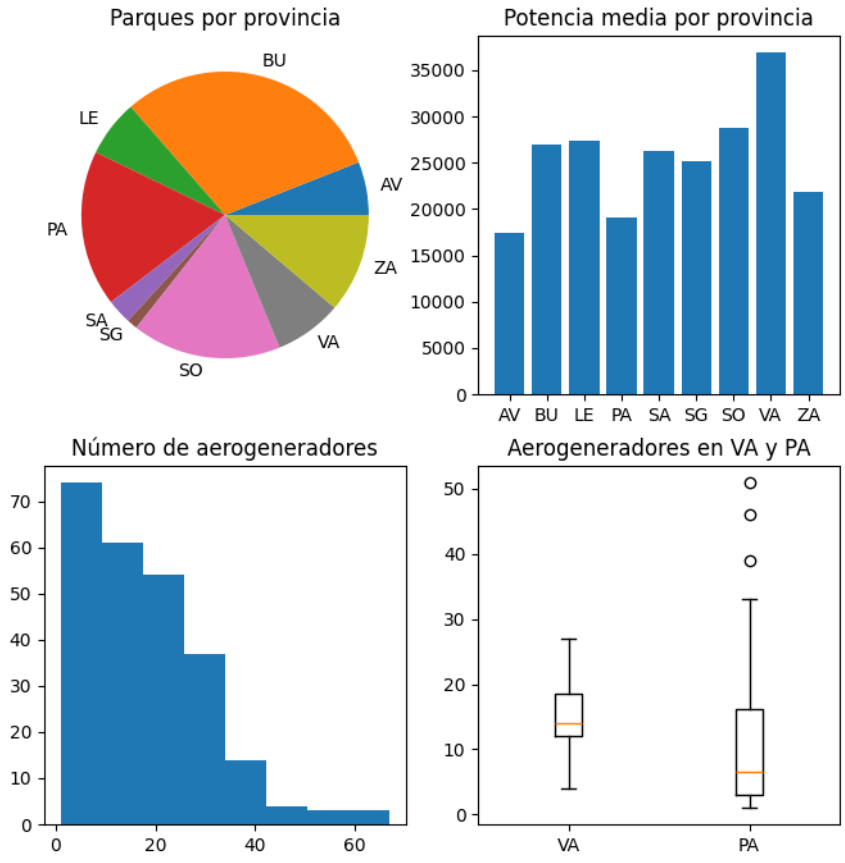
Nota: estas opciones deben ajustarse antes de mostrar el gráfico (antes de invocar al método `show`).

Ejercicio 5

Crea un programa **MultiplesGraficos.py** con un panel 2x2 para mostrar 4 gráficos diferentes sobre el dataset parques_eolicos.csv. En concreto se piden los siguientes gráficos:

- Gráfico circular que muestre la proporción de parques eólicos por provincia
- Gráfico de barras con la potencia media por provincia
- Histograma con la frecuencia del número de aerogeneradores
- Gráfico de cajas que muestre la dispersión del número de aerogeneradores en Valladolid y Palencia

Al final tendrás que obtener un resultado como éste:



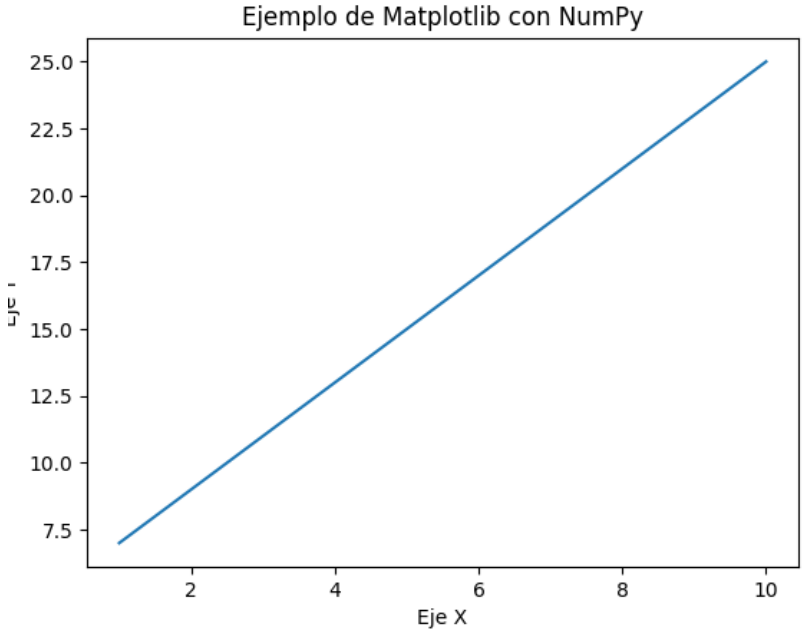
1.3. Uso combinado de *Matplotlib* y otras librerías

Matplotlib está integrada en otras librerías habituales de tratamiento de datos, de modo que resulta sencillo definir gráficos a partir de dichas librerías. Veremos un par de ejmplos.

1.3.1. *Matplotlib* y *NumPy*

Podemos unir las fuerzas de estos dos módulos que hemos visto, y representar gráficamente la información de un conjunto de datos. Por ejemplo, podríamos representar los valores de una función:

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

x = np.arange(1,11)
y = 2 * x + 5
plt.title("Ejemplo de Matplotlib con NumPy")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.plot(x,y)
plt.show()
```



1.3.2 *Pandas* y *Matplotlib*

Pandas tiene una integración inmediata con la librería *Matplotlib* para poder generar fácilmente gráficos a partir de las series o tablas obtenidas. Dentro de cada objeto de serie o *data frame* tenemos disponible una propiedad `plot` que permite dibujar diferentes tipos de gráficos. Por ejemplo:

```
import matplotlib.pyplot as plt
import pandas as pd

datos = pd.DataFrame(...)
```

```
# Gráfico de barras con los datos almacenados
datos.plot.bar()
plt.xlabel("Etiqueta eje X")
plt.ylabel("Etiqueta eje Y")
plt.title("Título del gráfico")
plt.show()
```

Como vemos, la propiedad `plot` permite generar el gráfico directamente con los datos del *data frame*, y luego desde el elemento `plt` podemos configurar otras opciones, como rótulos, leyendas, etc, antes de mostrar el gráfico. Podéis consultar más información sobre cómo generar estos gráficos aquí.

2. *Seaborn*

Seaborn es una extensión de *Matplotlib* para realizar otros gráficos más sofisticados, como por ejemplo mapas de correlación o mapas de clústeres. Aquí tenemos su web oficial. Para empezar, instalaremos la librería con el correspondiente comando `pip`:

```
pip install seaborn
```

Advertencia: igual que en el caso de *matplotlib*, esta librería ya deberías tenerla instalada. A la hora de importarla en nuestro código es habitual usar el alias o abreviatura `sns`, y utilizarla junto al núcleo de *Matplotlib* si es necesario:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

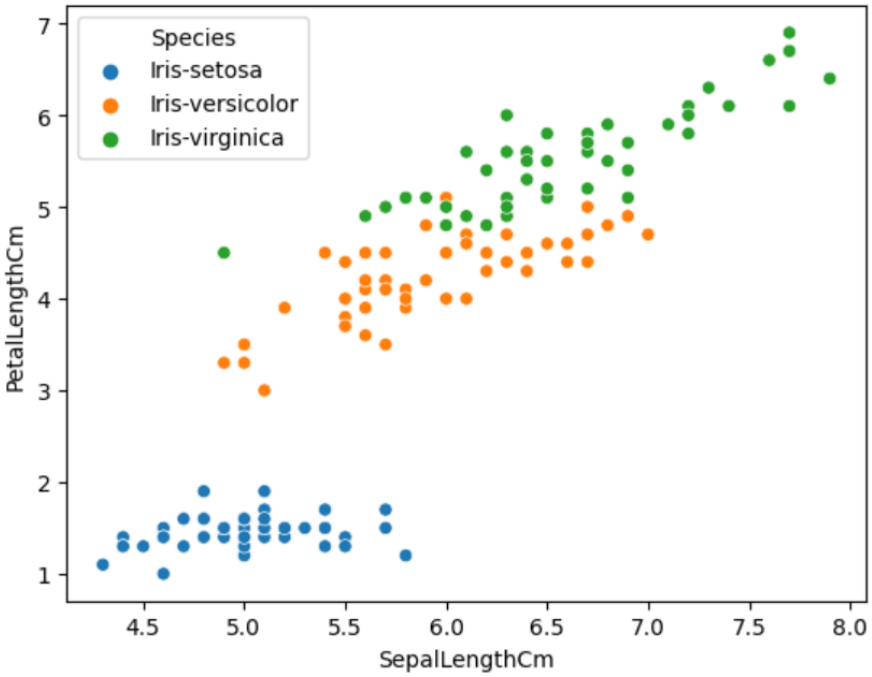
Para ilustrar las opciones que ofrece esta librería vamos a trabajar sobre un conjunto de datos ya definido. En concreto, usaremos el *dataset iris*, que comprende medidas y clasificaciones de diferentes tipos de la flor Iris. Podemos consultarlo aquí.

2.1. Gráficos simples

Con *Seaborn* podemos realizar gráficos similares a los que hacemos con *Matplotlib*, aunque con un estilo más particular. Así por ejemplo podemos hacer un diagrama de dispersión (*scatter plot*) que relacione la longitud de los sépalos con la de los pétalos en estas flores, y pinte de un color diferente los puntos de cada categoría de flor (columna *Species*):

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

datos = pd.read_csv('Iris.csv')
sns.scatterplot(x="SepalLengthCm", y="PetalLengthCm", data=datos,
hue="Species")
plt.show()
```

El resultado que obtenemos es éste:



En este caso puede verse con claridad que los tamaños mayores corresponden a la especie *virginica*, por ejemplo.

Podemos también contar cuántas flores hay de cada especie en el *dataset* usando un *count plot*:

```
sns.countplot(x="Species", data=datos)
```

Nota

En este caso concreto obtendríamos un gráfico con tres barras idénticas, ya que el *dataset* tiene el mismo número de muestras de cada especie.

2.2. Gráficos especiales

Sin embargo, la principal ventaja que podemos obtener empleando *Seaborn* es la de realizar gráficos que no están disponibles en la base proporcionada por *Matplotlib*. Veremos a continuación algunos de estos gráficos.

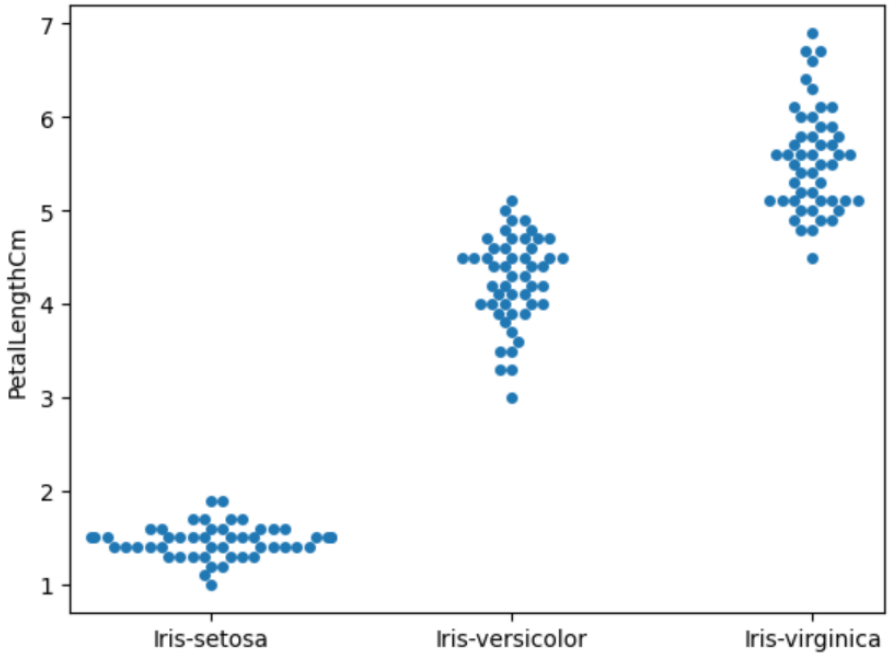
2.2.1. *Swarm plots*

Los *swarm plots* (que podríamos traducir al castellano como *gráficos de enjambre*) representan con puntos individuales las distintas observaciones de un conjunto de datos, pero agrupándolos de forma que parecen enjambres de insectos.

Por ejemplo, podemos ver la distribución de valores de las diferentes longitudes de pétalos para cada especie:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

datos = pd.read_csv('Iris.csv')
sns.swarmplot(x="Species", y="PetalLengthCm", data=datos)
plt.show()
```

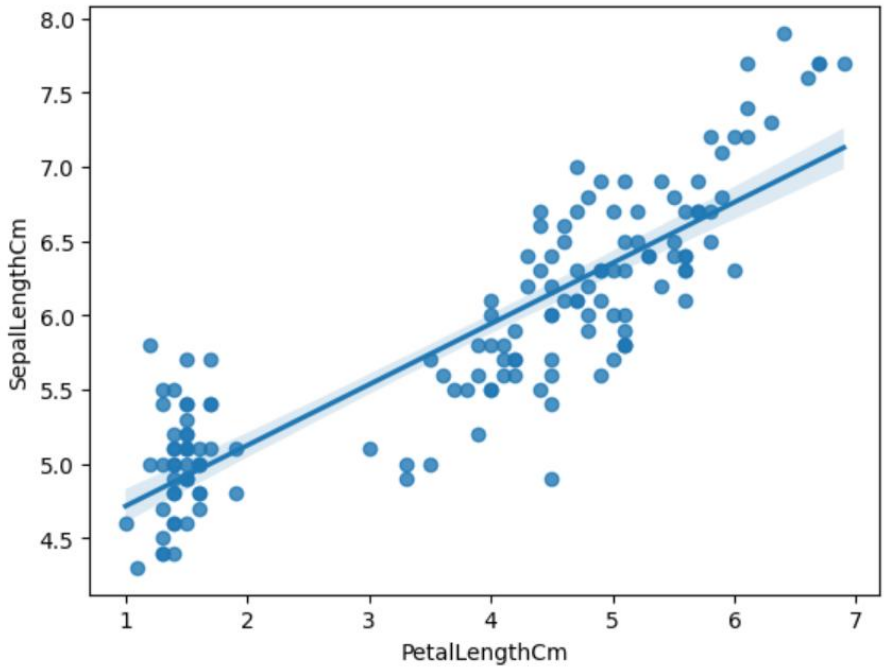
Obtendremos este resultado, donde podemos ver las longitudes habituales para cada categoría:



2.2.2. Gráficos de regresión

Los gráficos de regresión son algo similar a los de dispersión (*scatter plots*) pero permiten definir también la línea de regresión que ayuda a predecir el valor dependiente (Y) a partir de un determinado valor de X. Por ejemplo, podemos establecer una correspondencia entre la longitud del sépalo y la del pétalo para las flores anteriores:

```
# ... Mismo código de ejemplos anteriores
sns.regplot(x="PetalLengthCm", y="SepalLengthCm", data=datos)
plt.show()
```



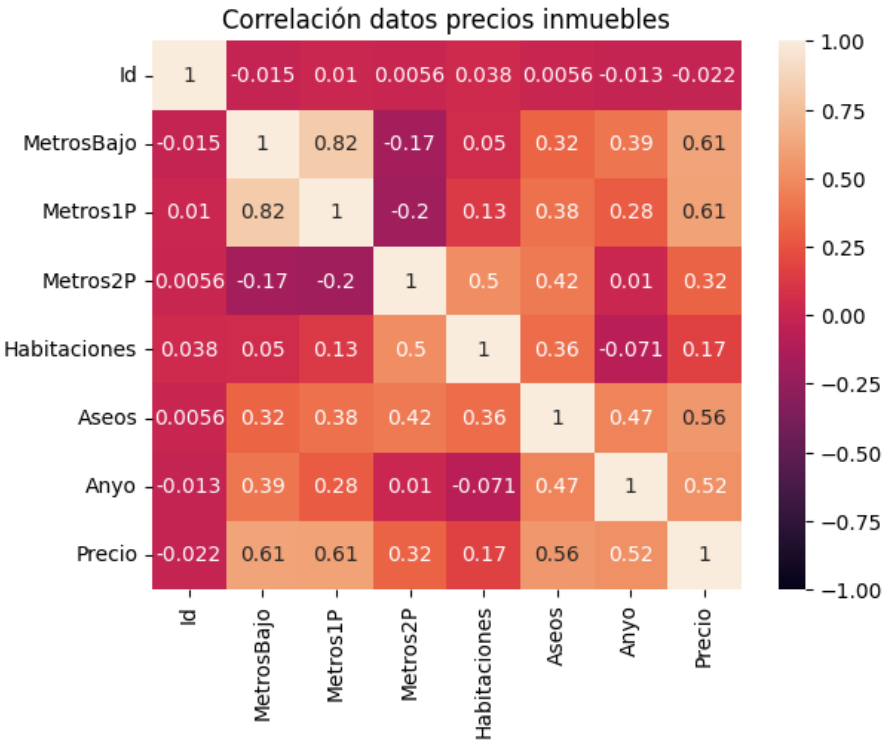
2.2.3. Mapas de correlación (*heatmaps*)

Los mapas de correlación (o mapas de calor, *heatmaps*) muestran una matriz de colores donde, visualmente, se puede establecer la correlación o dependencia entre diferentes campos de un *dataset*. Para ilustrar este concepto usaremos el dataset precios_inmuebles.csv que almacena datos de inmuebles en venta: metros cuadrados de las distintas plantas, año de construcción, número de aseos... y precio de venta. Vamos a establecer un mapa de correlación para ver cómo de correlacionadas están unas variables con otras.

Podemos establecer la correlación usando el método `corr` de pandas, y luego representar gráficamente esos datos usando el `heatmap` de Seaborn:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

datos = pd.read_csv('precios_inmuebles.csv')
heatmap = sns.heatmap(datos.corr(), vmin=-1, vmax=1, annot=True)
heatmap.set_title('Correlación datos precios inmuebles')
plt.show()
```

Obtendremos un gráfico como éste:

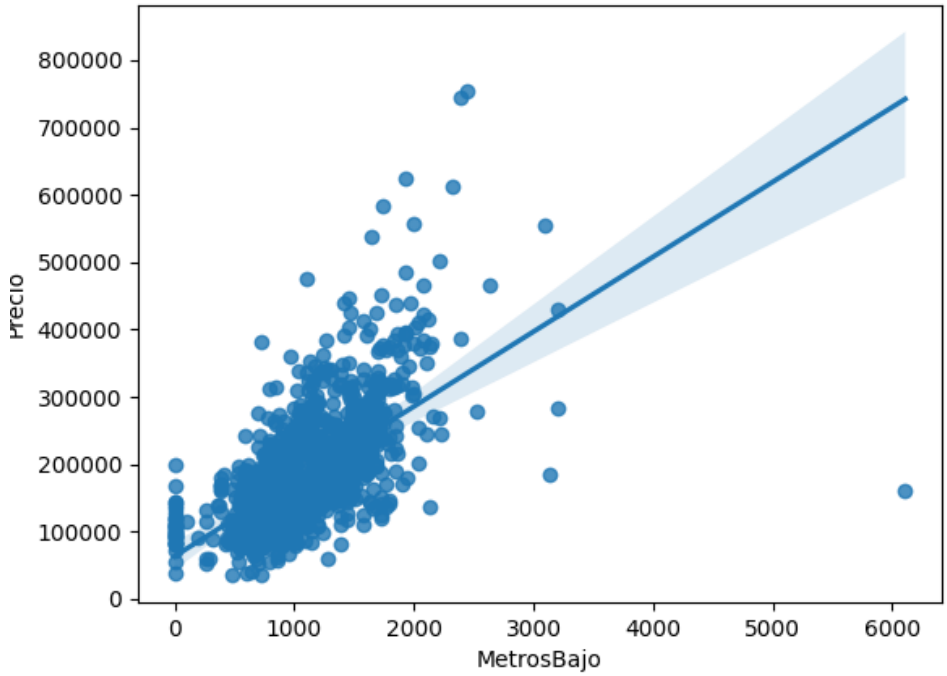


Podemos ver con la escala de colores y los valores numéricos los elementos más correlacionados. En la columna para el *Precio* vemos que los dos valores que más le afectan son los metros cuadrados de la planta baja y la primera planta. A su vez, si vamos a la columna *Metros1P* vemos que está muy correlacionada (0.82) con los *MetrosBajo*, con lo que podríamos prescindir de uno de los dos valores y quedarnos con el otro.

También podemos apreciar correlaciones inversas (negativas), aunque en este caso no son significativas. Indicarían que cuanto más sube el valor de una columna más baja el de la otra. En este caso podríamos intepretarlo suponiendo que las casas que tienen mucha superficie en las plantas baja y primera no suelen tener una segunda planta.

Ejercicio 6

Crea un programa **EstimaciónVivienda.py** que utilice el mismo CSV de inmuebles del ejemplo anterior para construir un mapa de regresión que ayude a estimar el precio final de la vivienda en base a los metros cuadrados de la planta baja. Deberá quedarte algo así:

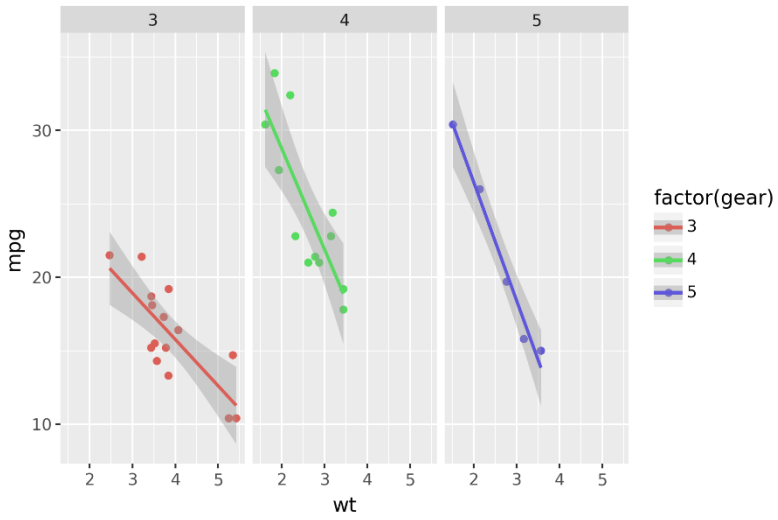

800000
700000
600000
500000
400000
300000
200000
100000
0
Precio
0
1000
2000
3000
4000
5000
6000
MetrosBajo

3. Otras librerías alternativas

Además de *Matplotlib* y *Seaborn*, existen otras librerías que podemos utilizar para realizar representaciones gráficas de datos. Aquí indicamos algunas de ellas:

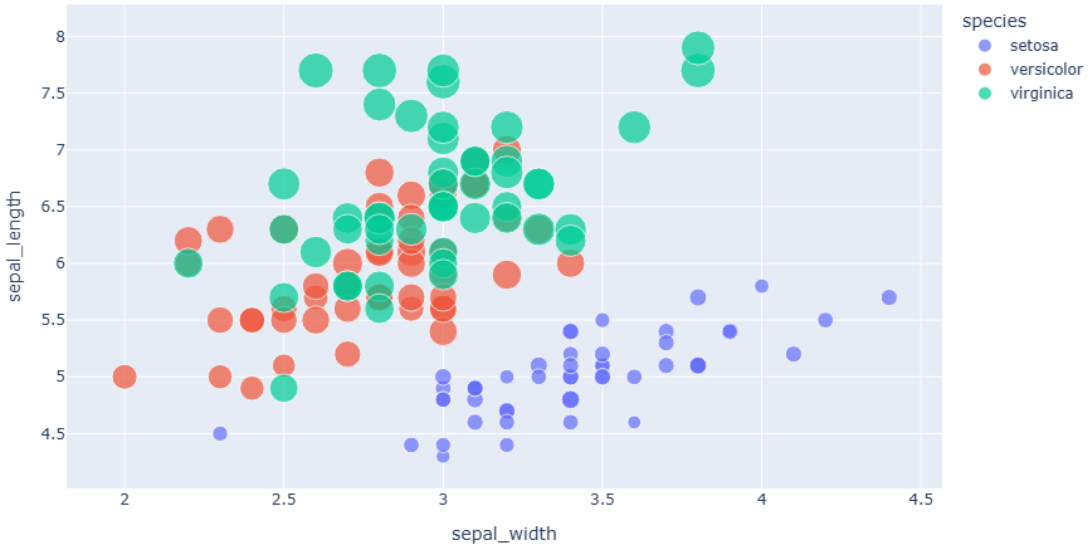
- **Plotnine**: librería de reciente creación, con características similares a *Matplotlib*, aunque no tan extendida. Veamos un ejemplo:

```
from plotnine import ggplot, geom_point, aes, stat_smooth, facet_wrap
from plotnine.data import mtcars

(
    ggplot(mtcars, aes("wt", "mpg", color="factor(gear)"))
    + geom_point()
    + stat_smooth(method="lm")
    + facet_wrap("gear")
)
```



- **Plotly**: permite definir gráficos interactivos, donde podemos modificar datos o filtrar sobre el propio gráfico. Veamos un ejemplo:

```
import plotly.express as px

df = px.data.iris()

fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species",

                 size='petal_length', hover_data=['petal_width'])

fig.show()
```



Nota: este gráfico realmente es interactivo, aunque en la imagen no pueda probarse.