**Clase 10:** **Visualización de Datos I**

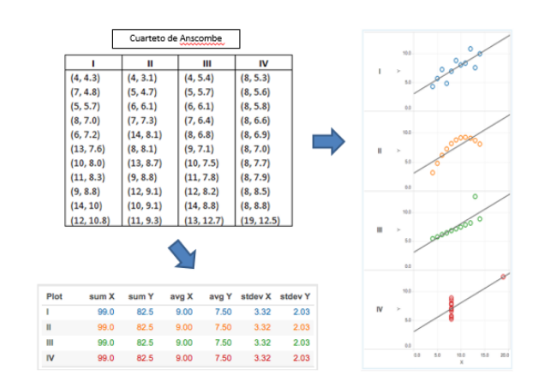
La **visualización** es un **medio útil** para **examinar, entender y transmitir información.** Permite **explorar los datos**; **expresar relaciones complejas**, **condensar información** y **comunicarla** de manera **más eficaz**.

Los **Sistemas de Visualización** basados en computadora dan representaciones de datos diseñados para **ayudar a las personas** a realizar **tareas de forma más eficaz** (Ejemplo, Blue Yonder).

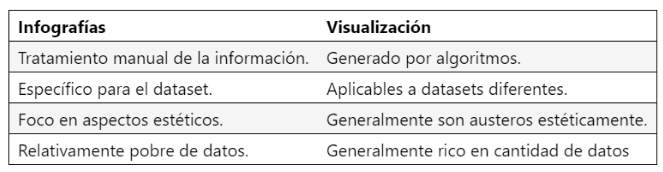
Ayuda cuando en lugar de reemplazar a las personas con métodos computacionales de toma de decisiones, se necesita **aumentar las capacidades humanas**.

La **visualización de datos** definitivamente **está orientada a las personas**.

Ejemplo: 4 datasets con iguales parámetros, podrían dar lugar a decisiones equivocadas analizándolos solo en tablas. Con gráficos se puede ver que son bien distintos, pese a compartir parámetros.



**Infografías:** **Combinación de textos, íconos, mapas y gráficos** para **comunicar** información **en forma visual**. Lo que busca es producir **IMPACTO**. No es lo mismo que visualización:



**Por qué** el usuario necesita los datos, **qué datos** deben mostrarse y en **qué formato (diseño)**. Para esto hay que tener en cuenta 3 limitantes de recursos:

* **Limites de las Computadoras**
* **Límites de los Humanos**
* **Límites de las Pantallas**

**Atributos de los Elementos Gráficos**

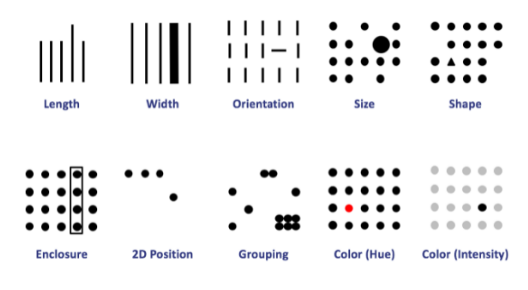
Entre otros son: Tamaño, forma, color, textura, transparencia, posición, orientación, alineación, movimiento (en algunos casos). Cada tipo de gráfico usa uno o más. Algunos impactan más que otros:



Percepción Visual: Según cómo se presenta la información, es más fácil o más difícil interpretarla:



Hay varios recursos visuales que se pueden usar para transmitir información:

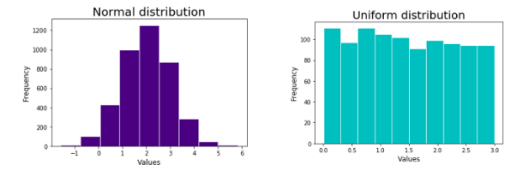


Longitud, ancho, orientación, tamaño, forma, segregación, posición en 2D; agrupamiento, color, intensidad de color.

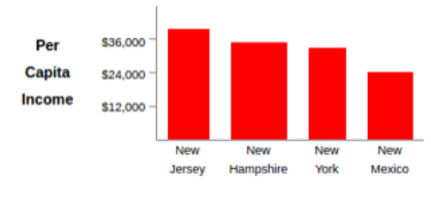
**Tipos de Gráficos**

Los más usados son:

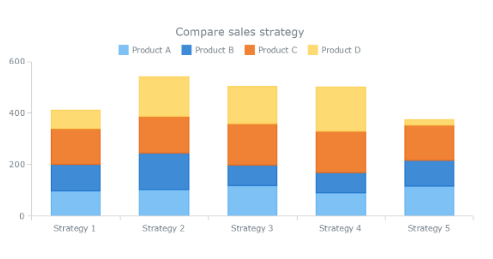
* **Histogramas**: Podemos ver cómo y en qué valores se concentran los datos. Puede ayudarnos a interpretar qué distribución se podría ajustar mejor a nuestros datos.



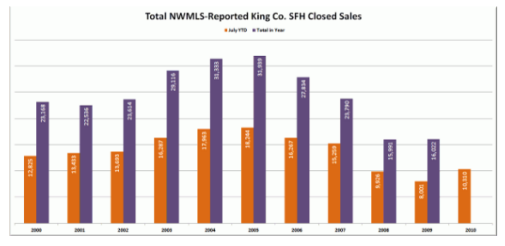
* **Barras** (bar Chart): Sirve para mostrar datos numéricos separables en diferentes categorías.



* **Barras Apiladas** (Stacked bar chart): Ayudan a comparer distribuciones de distintas poblaciones o series de elementos.



* **Gráfico de barras (2 o más variables):** Ayuda a visualizar niveles y distribución relativa de cada variable, así como también compararlas entre sí.



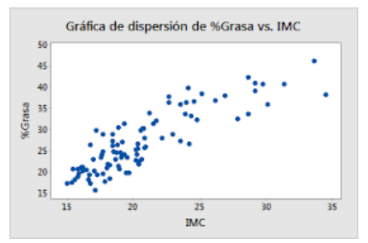
* **Tortas** (pie Chart): Se pueden ver porporciones relativas o porcentajes de una variable.



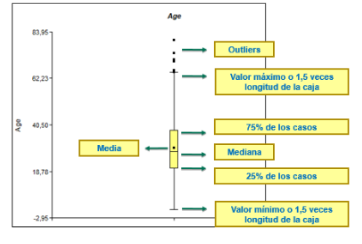
* **Gráfico de líneas** (plot): Se puede analizar la relación entre 2 o más variables; generalmente se usan para estudiar la evolución en el tiempo (eje X) de una o más variables (eje Y), analizar picos, valles, detectar patrones; etc.



* **Dispersión** (Scatterplot): Ayudan a detectar tendencias, concentración de casos y outliers.



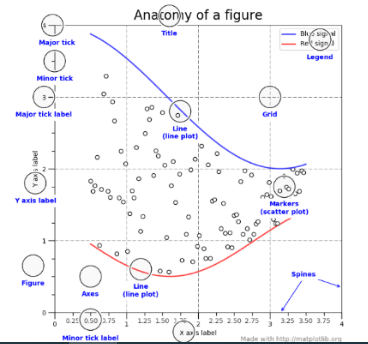
* **Diagramas de caja** (Box Plot): Ayuda a representar gráficamente una serie de datos estadísticos de una variable. Se puede ver a simple vista la **media**, la **mediana**, los **cuartiles** y los **outliers**.



**Matplotlib:** Biblioteca para visualización de datos construida sobre Numpy arrays y diseñada para trabajar con todo el ecosistema **ScyPy**. 20 años de historia. Muchos ejemplos y tutoriales. Funciona bien con muchos sistemas operativos. A partir de Matplotlib se crearon muchas librerías nuevas que facilitan y especializan su forma de graficar, tales como Seaborn, gplot, etc.

Sitio útil para usar matplotlib: <https://python-graph-gallery.com/cheat-sheets/>

Acá se pueden ver los principales elementos de los gráficos que usa matplotlib:



Matplotlib tiene varios estilos estándar predefinidos. Los mismos pueden verse con **plt.style.available**

Para definir algún estilo específico, por ejemplo classic, escribimos **plt.style.use(‘classic’)**

**Figure**/Figura: Es como un contenedor de todos los objetos que representan el gráfico.

Una figura contiene uno o más **axes** (áreas independientes donde generamos los gráficos; son los ejes).

Hay dos formas de visualizar los gráficos creados con matplotlib:

1. **%matplotlib inline** muestra los gráficos debajo de la celda de código ejecutada y se ejecuta sólo una vez por kernel/sesión. Una vez ejecutado, cualquier celda de la notebook que cree un gráfico, mostrará debajo la imagen resultado.
2. **plt.show()** muestra el último gráfico creado, independientemente de la ubicación de la celda.

IE:

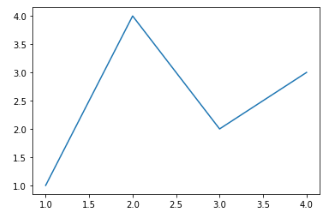
Import matplotlib as plt

%matplotlib inline

fig = plt.figure() acá está generando una figura.

ax = plt.axes() acá está generando un eje.

ax.plot([1, 2, 3, 4][1, 4, 2, 3]) acá está dibujando una línea; primero recibe los valores del eje X, luego los valores correspondientes para el eje Y:



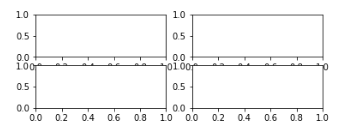
Con el método **plt.subplot** se pueden crear simultáneamente la figura y los axes.

IE:

%matplotlib inline

fig, axs = plt.subplots(2, 2, figsize=(6,2))

En este caso creó la figura y los axes simultáneamente, indicando que en la figura quiere una tabla de 2 x 2 ejes, los cuales deben tener un tamaño de 6 (eje x) por 2 (eje y):



El método **plt.grid()** permite crear grillas sobre los ejes. Una alternativa, para generar una grilla vacía es **plt.style.use(‘seaborn-whitegrid’)**

Es posible referenciar cada uno de los ejes:

Axe [0,0]: Es el de arriba a la izquierda

Axe [0,1]: Es el de arriba a la derecha

Axe [1,0]: Es el de anajo a la izquierda

Si el usar el método subplots defino sharex = True y sharey = True, lo que estoy indicándole es que todos los ejes comparten las escalas de x e y

IE:

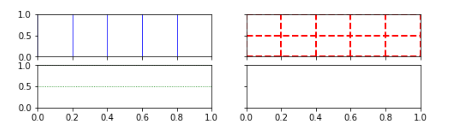
fig, axs = plt.subplots(2,2, figsize=(8,2), sharex = True, sharey = True)

axs[0,0].grid(axis=’x’, color=’blue’)

axs[0,1].grid(color=’r’, linestyle=’--‘, linewidth=2)

axs[1,0].grid(axis=’y’, color=’g’, linestyle=’:’)

plt.show()



**Hay diferentes alternativas para especificar Color:**

* color = ‘blue’ Se especifica el color por su nombre
* color = ‘g’ código de color corto (rgbcmyk)
* color = 0.75 (es una escala de grises entre 0 y 1)
* color = ‘#FFDD44’ código Hexadecimal /RRGGBB) desde 00 hasta FF
* color = (1.0, 0.2, 0.3) tupla RGB con valores para cada uno entre 0 y 1

**Hay diferentes alternativas para especificar Estilos de línea:**

* linestyle = ‘solid’ o bien linestyle = ‘-’ es una línea continua
* linestyle = ‘dashed’ o bien linestyle = ‘--’ es una línea de trazos
* linestyle = ‘dashedot’ o bien linestyle = ‘-.’ es una línea compuesta (punto y segmento)
* linestyle = ‘dotted’ o bien linestyle = ‘:’ es una línea punteada

**Existen también combinaciones para especificar Color y Estilos de línea simultáneamente:**

* ‘-g’ línea contínua verde.
* ‘--c’ línea de guiones cyan.
* ‘-.k’ línea compuesta negra.
* ‘:r’ línea punteada roja.

Con el método **figure.suptitle** es posible ponerle un Título a la figura.

Con el método **axes.set** es posible ponerles un título a los ejes (**xlabel** y **ylabel**), así como también al plot (**title**).

IE:

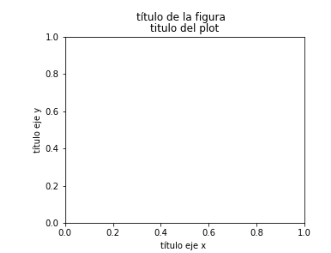
fig = plt.figure(figsize=(5,4))

ax = plt.axes()

fig.suptitle(“título de la figura”)

ax.set(xlabel = ”título eje x”, ylabel = ”título eje y”, title = “título del plot”)

plt.show()



Si creamos un gráfico de líneas con **plot**, podemos indicar el nombre de cada serie con **label**. También podemos generar una leyenda en el gráfico que indique esos nombres con **legend**. Se puede indicar el lugar donde colocar la leyenda (**loc**) y el título (**title**). Con **frameon=True** como argumento en la leyenda, podemos dibujar una caja alrededor de la leyenda.

**Valores posibles de ubicación de leyenda usando el argumento loc:**

* ‘best’
* ‘upper right’
* ‘upper left’
* ‘lower right’
* ‘lower left’
* ‘right’
* ‘center left’
* ‘center right’
* ‘lower center’
* ‘upper center’
* ‘center’

IE:

import numpy.random as rd

rd.seed(0)

random1 = rd.randn(30).cumsum()

random2 = rd.randn(30).cumsum()

fig = plt.figure(figsize=(6,2))

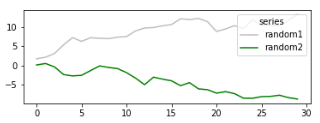
ax = plt.axes()

ax.plot(random1, color = ‘0.75’, label=’random1’)

ax.plot(random2, color = ‘g’, label=’random2’)

ax.legend(loc=’upper right’, title = ‘series’)

plt.show()



Podemos indicar los intervalos en los cuales queremos graficar los ejes x e y con los métodos **set\_xlim** y **set\_ylim**. En lugar de definir los límites con una tupla de mínimo y máximo, podemos pedirle al sistema que los ajuste a los valores de los datos con la sentencia **ax.axis(tight)**

IE:

fig = plt.figure(figsize=(6,2))

ax = plt.axes()

ax.set\_xlim(5,12)

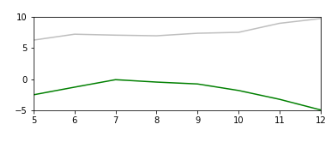
ax.set\_ylim(-5,10)

# ax.set(xlim = (5,12), ylim = (-5,10)) hubiera sido otra forma de definer lo mismo.

ax.plot(random1, color = ‘0.75’, label=’random1’)

ax.plot(random2, color = ‘g’, label=’random2’)

plt.show()



**Gráfico de Líneas:** Podemos crearlo con el método **.plot**

Ya vimos cómo cambiar el color y tipo de las líneas y cómo agregarles un label (etiqueta) más arriba. También es posible destacar los puntos de las series con **marker**; hay una manera alternativa de definir el color y la línea (**k--**). Se pueden modificar los valores a mostrar del eje X y/o Y con **set\_xticks** o **set\_yticks** y también pueden modificarse los labels con **set\_xticklabels** y **set\_yticklabels**.

IE:

rd.seed(10)

random1 = rd.random(30).cumsum()

fig = plt.figure(figsize=(8,4))

ax = plt.axes()

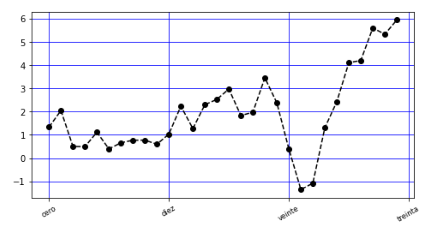
ax.set.xticks([0, 10, 20, 30])

ax.set.sticklabels([‘cero’, ‘diez’, ‘veinte’, ‘treinta’],rotation=30, fontsize=’small’)

ax.grid(color=’b’)

ax.plot(random1, ‘k--‘,marker=’o’)

plt.show()



**Histogramas:** Se pueden graficar usando el método **.hist** Se pueden establecer varios parámetros, tales como:

* **Bins:** entero, lista o string.

Si se trata de un entero, define la cantidad de bins (barras) de igual ancho en el rango de valores del gráfico.

En caso de ser una lista, define los límites de cada barra incluyendo el límite izquierdo del primer bin y el límite derecho del último bin. Pueden ser de anchos distintos.

Si es un string, representa una de las estrategias que usa numpy.histogram\_bin\_edges: ‘auto’, ‘fd’, ‘doane’, ‘scott’, ‘stone’, ‘rice’, ‘sturges’, ‘sqrt’.

* **Density:** Boolean. Si True, los pesos están normalizados de forma tal que la integral de densidad en el rango vale 1.
* **Histtype:** String. Tipo de histograma a graficar entre ‘bar’ (histograma tradicional), ‘barstacked’ (histograma de barras apiladas), ‘step’ (gráfico de líneas, por default sin relleno) y ‘stepfilled’ (gráfico de líneas con relleno por default).
* **Color:** Que tomarán las barras del histograma.
* **Edgecolor:** Color de los bordes de las barras del histograma.
* **Alpha:** decimal de valor entre 0 y 1 que representa el nivel de transparencia (0 totalmente transparente – 1 totalmente opaco).

En caso de tener decidido usar los mismos valores de parámetros para un conjunto de parámetros a aplicar en diferentes variables, conviene crear un diccionario con los mismos y agregar dicho diccionario al insertar el código de gráfico de cada variable y pasarlo como parámetro con \*\*nombrediccionario. De esta forma también permitimos la flexibilidad de que la información esté o no esté, y el gráfico va a salir igual.

IE:

rd.seed(10)

random1 = rd.randn(30).cumsum()

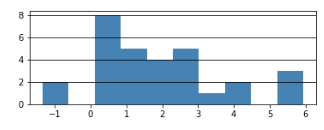
fig = plt.figure(figsize=(6,2))

ax = plt.axes()

ax.hist(random1, color=’steelblue’)

ax.grid(axis=’y’, color=’k’)

plt.show()



**Gráfico de Barras**: El diagrama de una sola barra se puede hacer con el método **.bar** colocando en el **primer parámetro** la **posición** de cada valor de la serie **en el eje x**; como **segundo parámetro** se coloca la **serie**, como **tercer parámetro** se coloca el **ancho de la serie**. Se puede agregar más parámetros.

IE:

import numpy as np

import matplotlib.pyplot as plt

womenMeans = (25, 32, 34, 20, 25)

ind = np.arange(5) #estas van a ser las posiciones en x

width = 0.35

fig = plt.figure(figsize=(6,2))

ax = plt.axes()

ax.bar(ind, womenMeans, width, color=’r’)

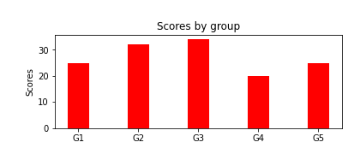
ax.set\_ylabel(‘Scores’)

ax.set\_title(‘Scores by group’)

ax.set\_xticks(ind)

ax.set\_xticklabels([‘G1’, ‘G2’, ‘G3’, ‘G4’, ‘G5’])

plt.show()



**Diagrama de Barras Apiladas:** Se genera con el método **.bar** usando el parámetro **bottom** para indicar la barra sobre la cual se apila.

IE:

menMeans = (20, 35, 30, 35, 27)

womenMeans = (25, 32, 34, 20, 25)

ind = np.arange(5)

width = 0.35

fig = plt.figure(figsize=(6,2))

ax = plt.axes()

ax.bar(ind, menMeans, width, color=’r’)

ax.bar(ind, womenMeans, width, color=’b’, bottom=menMeans)

ax.set\_ylabel(‘Scores’)

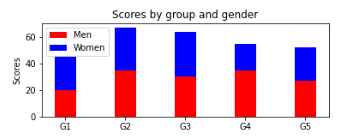
ax.set\_title(‘Scores by group and gender’)

ax.set\_xticks(ind)

ax.set\_xticklabels([‘G1’, ‘G2’, ‘G3’, ‘G4’, ‘G5’])

ax.legend(labels=[‘Men’,’Women’])

plt.show()



**Diagrama de Barras Varias Variables**: Se hace con el método **.bar**, modificando la posición de cada barra en el eje x, para que no se superpongan. Se recomienda que tengan el mismo ancho de barras.

IE:

menMeans = (20, 35, 30, 35, 27)

womenMeans = (25, 32, 34, 20, 25)

ind = np.arange(5)

width = 0.35

fig = plt.figure(figsize=(6,2))

ax = plt.axes()

ax.bar(ind+0.00, menMeans, width, color=’r’)

ax.bar(ind+0.15, womenMeans, width, color=’b’)

ax.set\_ylabel(‘Scores’)

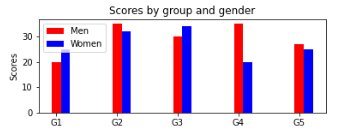
ax.set\_title(‘Scores by group and gender’)

ax.set\_xticks(ind)

ax.set\_xticklabels([‘G1’, ‘G2’, ‘G3’, ‘G4’, ‘G5’])

ax.legend(labels=[‘Men’,’Women’])

plt.show()



**Gráfico de Tortas**: Sirve mucho para graficar una serie con pocos elementos. Cada porción de torta muestra la medida de cada número de la serie, proporcional al total de la suma de los ítems. Se puede crear con el método **.pie**

IE:

from matplotlib import pyplot as plt

import numpy as np

menMeans = (20, 35, 30, 35, 27)

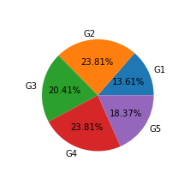
groups = [‘G1’, ‘G2’, ‘G3’, ‘G4’, ‘G5’]

fig = plt.figure(figsize=(5,3))

ax = plt.axes()

ax.pie(menMeans, labels = groups, autopct=’%1.2f%’)

plt.show()



**Diagrama de Dispersión (Scatterplot)**: Se usa para mostrar cómo dos variables se relacionan entre sí. La intersección entre los valores de ambas series se representa con un punto. Puede representarse una tercer variable cambiando el color o la medida de los puntos. Se genera usando el método **.scatter** Cuenta con un argumento alfa que puede ponerse al final del ax.scatter con valor que puede variar entre 0 (transparente) y 1 (opaco). Pueden definirse diferentes markers como argumentos (por default es ‘o’, pero también podría usarse ‘s’, ‘+’, ‘\*’). También puede definirse color y label. El orden es datax; datay; color; label; alpha.

IE:

Girls = [89, 90, 70, 89, 100, 80, 90, 100, 80, 34]

Boys = [30, 29, 49, 48, 100, 48, 38, 45, 20, 30]

Promedio = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

fig = plt.figure(figsize=(5,2))

ax = plt.axes()

ax.scatter(promedio, girls, color=’r’)

ax.scatter(promedio, boys, color=’b’)

ax.set\_xlabel(‘Promedio’)

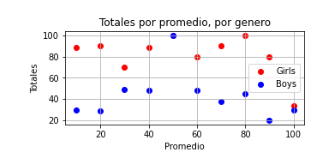
ax.set\_ylabel(‘Totales’)

ax.grid()

ax.set\_title(‘Totales por promedio, por género’)

ax.legend(labels=[‘Girls’,’Boys’])

plt.show()



**Librería Seaborn**

Seaborn da **comandos de más alto nivel** que Matplotlib para crear una **variedad de gráficos** útiles **para explorar datos estadísticos**. Tiene funciones de visualización que permiten expresar **más información de forma más efectiva**. Vamos a ver gráficos que también pueden crearse con matplotlib, pero la API de Seaborn es más amigable.

Import numpy as np

Import pandas as pd

Import seaborn as sns

**Todos los métodos de Seaborn** usan **parámetros similares:** **data** para el **dataset**, **x** e **y** para las **variables**.

Con el método **sns.scatterplot** se puede crear un scatterplot.

IE:

iris = sns.load\_dataset(‘iris’) iris es un DataFrame.

sns.set\_style(‘whitegrid’)

plt.figure(figsize=(5,2))

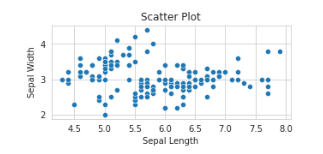
sns.scatterplot(data=iris, x=”sepal\_length”, y=”sepal\_width”)

plt.xlabel(“Sepal Length”)

plt.ylabel(“Sepal Width”)

plt.title(“Scatter Plot”)

plt.show()



Podemos generar un gráfico Box Plot con el método **sns.boxplot**

Con **sns.set\_style** se puede seleccionar un estilo Seaborn para el gráfico, tales como whitegrid o darkgrid.

Con el método **sns.load.dataset** podemos cargar un Dataset.

IE:

sns.set\_style(‘darkgrid’)

plt.figure(figsize=(5,2))

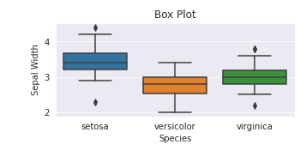
sns.boxplot(data=iris, x =”species”, y=”sepal\_width”)

plt.xlabel(“Species”)

plt.ylabel(“Sepal Width”)

plt.title(“Box Plot”)

plt.show()



Con la función **sns.displot** se puede crear una familia de gráficos, los cuales se indican en el parámetro **kind**. Por default, crea un histograma. Grafica una o dos variables, y es posible graficar una tercer variable usando el parámetro **hue**. En el siguiente ejemplo, se crea un histograma con el adicional de que pone las barras en forma apilada (con **multiple=”stack”**) y calculando la probabilidad por barra (con **stat=”probability”**)

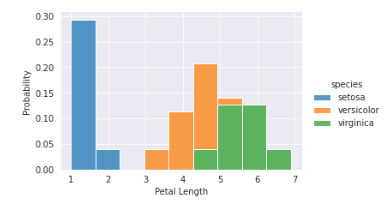
IE:

g = sns.displot(date=iris, x=”petal\_length”, hue=”species”, multiple=”stack”, stat=”probability”)

g.fig.set\_size\_inches(6,3)

g.set\_axis\_labels(“Petal Length”, “Probability”)

plt.show()



Con **sns.kdeplot** (**K**ernel **D**istribution **E**stimation **Plot**) se puede obtener una **estimación** **uniforme de la distribución** usando una **estimación de densidad kernel**. El parámetro que recibe sns.kdeplot es una serie. Es posible superponer varias series.

Con **shade=True** hacemos que se pinte el área debajo de la línea.

IE:

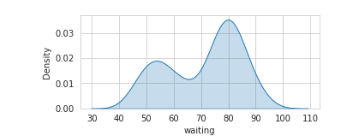
geyser = sns.load.dataset(“geyser”) geyser es un DataFrame

fig = plt.figure(figuresize=(5,2))

sns.set\_style(‘whitegrid’)

sns.kdeplot(data=geyser, x=’waiting’, shade=True)

plt.show()

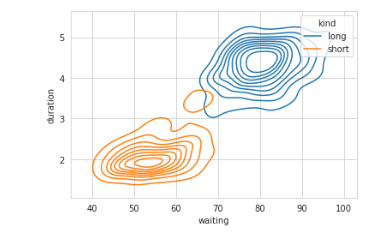


Al indicar parámetros x e y, es posible calcular la distribución de una variable respecto de otra. Es posible incluso representar una tercer variable, la cual se representará con un color distinto, usando el parámetro **hue**.

IE:

sns.kdeplot(data=geyser, x=”waiting”, y=”duration”, hue=”kind”)

plt.show()



**sns.jointplot** sirve para mostrar dos tipos de gráficos para dos variables. Por default da un scatterplot en el centro y la distribución en los costados, pero es posible que usar otros gráficos. Con el parámetro **hue** es posible involucrar una tercer variable también.

IE:

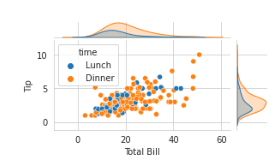
tips = sns.load.dataset(“tips”)

g = sns.jointplot(data=tips, x=’total\_bill’, y=’tip’, hue=’time’)

g.fig.set\_size\_inches(4,2)

g.set\_axis\_labels(“total\_bill, “Tip”)

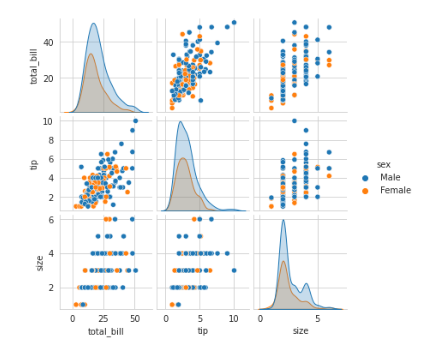
plt.show()



**Gráficos PairPlot**: Son muy útiles para explorar correlaciones entre datos multidimensionales, graficando todos los pares de valores entre sí. Se generan usando el método **sns.pairplot**

IE:

sns.pairplot(data=tips, hue=’sex’, height=1.8)



**Facetgrid:** Al igual que los PairPlot, son gráficos muy útiles para explorar la correlación entre datos multidimensionales, relacionando todos los pares de valores entre sí. Relaciona variables con los valores de una variable categórica. Se crean usando el método **sns.FacetGrid** Por ejemplo, vamos a usar un scatterplot entre las variables “total\_bill” y “tip”, haciendo gráficos distintos por cada valor distinto de la variable “time”.

IE:

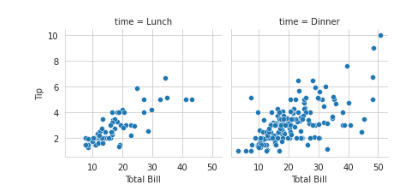
g = sns.FacetGrid(tips, col=”time”)

g.map(sns.scatterplot, “total\_bill”, “tip”)

g.fig.set\_size\_inches(6,3)

g.set\_axis\_labels(“Total Bill”, “Tip”)

plt.show()



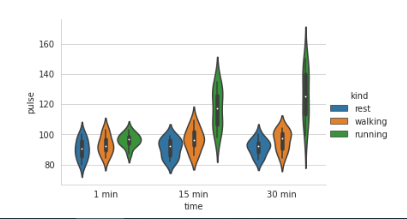
**Gráficos Catplot**: Muestran la relación entre una variable numérica y una o más variables categóricas. Catplot define una familia de gráficos, los cuales se indican en el parámetro **kind**. Se generan usando el método **sns.catplot**

IE:

exercise = sns.load\_dataset(“exercise”)

g = sns.catplot(x=”time”, y=”pulse”, hue=”kind”, data=exercise, kind=”violin”)

g.fig.set\_size\_inches(6,3)



**Mapas:** Son tipos de gráficos muy útiles para representar datos geoespaciales. En general disponemos de dos capas:

1. **Los datos que se quieren mostrar** (necesitamos coordenadas).
2. **El mapa donde se visualizan esos datos**.

Geopandas extiende los métodos de matplotlib y, con la librería **contextility**, ofrece mapas donde pueden graficarse los datos. Por ejemplo, se pueden graficar pozos petroleros de Argentina sobre un mapa del país:

IE:

df\_pozos = pd.read\_csv(“../Data/capitulo-iv-pozos.csv” )

Se transforman las coordenadas numéricas en la forma geométrica Punto con gpd.points\_from\_xy:

geo\_pozos = dps.GeoDataFrame(df\_pozos, geometry= gpd.points\_from\_xy(df\_pozos.coordenadax, df\_pozos.coordenaday))

Entonces ahora tenemos un GeoDataFrame en vez de un DataFrame.

Los pozos están ubicados a partir de sus coordenadas de latitud y longitud.

fig, ax = plt.subplots(figsize=(4,6))

ax.set\_title(‘Pozos’)

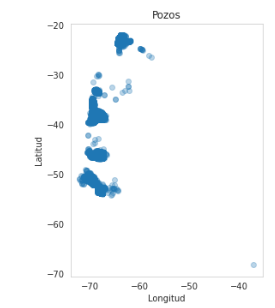
ax.set\_xlabel(‘Longitud’)

ax.set\_ylabel(‘Latitud’)

plt.grid()

geo\_pozos.plot(ax=ax, legend=True, alpha=0.3)

plt.show()



Tenemos los pozos, pero no tenemos el mapa de Argentina. Vamos a agregarlo entonces. Para esto tenemos que informar qué sistema de coordenadas (CRS) estamos usando (EPSG:4326 es latitud y longitud). Como la librería contextitlity exige sistema CRS 3857 (Spheric Mercator); necesitamos cambiar a este CRS para poder usarla. Esto podemos hacerlo con el método **.to\_crs** Por último, con el método **add\_basemap** de la librería **contextility**, podemos finalmente agregar el mapa.

IE:

fig, ax = plt.subplots(figsize=(4,6))

ax.set\_title(‘Pozos’)

ax.set\_xlabel(‘x – Spheric Mercator’)

ax.set\_ylabel(‘y – Spheric Mercator’)

plt.grid()

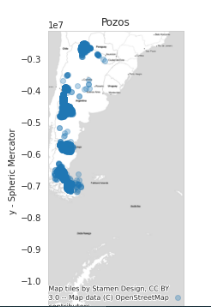
geo\_pozos.crs = “EPSG:4326” #Acá estoy indicando de qué sistema de coordenadas vengo

geo\_pozos\_3857 = geo\_pozos.to\_crs(epsg=3857)

#Acá hice el cambio de coordenadas para poder acceder a los mapas de contextility

geo\_pozos\_3857.plot(ax = ax, legend = True, alpha = 0.3)

ctx.add\_basemap(ax, source=ctx.providers.Stamen.TonerLite)



Es possible hacer un zoom en el mapa para ver con más detalle una parte del mismo. Para esto podemos usar el método **bounds**, que nos permite hacer un slice de los puntos a partir de sus coordenadas.

IE:

geo\_pozos\_norte = geo\_pozos[(geo\_pozos.bound.minx<-60) & (geo\_pozos.bound.maxx>-65) &(geo\_pozos.bound.miny<-20) & (geo\_pozos.bound.maxy>-30)]

Repitiendo el proceso podemos rehacer el mapa con este zoom:

fig, ax = plt.subplots(figsize=(4,6))

ax.set\_title(‘Pozos del norte Argentino’)

ax.set\_xlabel(‘x – Spheric Mercator’)

ax.set\_ylabel(‘y – Spheric Mercator’)

plt.grid()

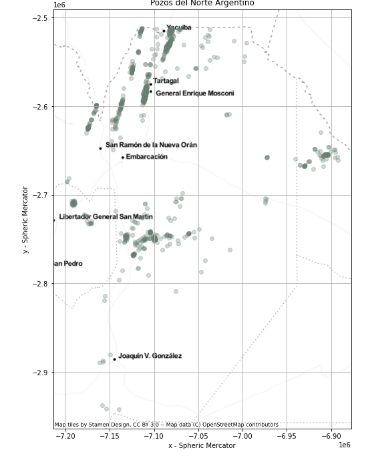
geo\_pozos\_norte.crs = “EPSG:4326” #Acá estoy indicando de qué sistema de coordenadas vengo

geo\_pozos\_norte\_3857 = geo\_pozos\_norte.to\_crs(epsg=3857)

#Acá hice el cambio de coordenadas para poder acceder a los mapas de contextility

geo\_pozos\_norte\_3857.plot(ax = ax, legend = True, alpha = 0.3)

ctx.add\_basemap(ax, source=ctx.providers.Stamen.TonerLite)



Es posible generar mapas con matplotlib usando el método **ax.scatter** para marcar los puntos y el método **imshow** para graficar la imagen con el mapa.

Por ejemplo, podemos graficar puntos de Rio de Janeiro en un mapa de la ciudad:

IE:

data = {‘longitud’: [-22.984, -22.975, -22.958, -22.969], ‘latitud’:[-43.197, -43.183, -43.209, -43.227], ‘lugar’=[‘Jardín Botánico’, ‘Cristo Redentor’, ‘Praia Copacabana’, ‘Praia Ipanema’]}

df\_rio = pd.DataFrame(data)

Se genera un rectángulo con valores de latitud y longitud tales que puedan contener a todos los lugares:

BBox = (-22.9898, -22.94614, -43.2337, -43.1680)

Se carga el mapa generado con OpenStreetMap. Para ver cómo, ir a:

<https://medium.com/@abuqassim115/thanks-for-your-response-frank-fb869824ede2>

ruh\_m = plt.imread(../Data/map.png)

fig, ax = plt.subplots(figsize=(8,7))

ax.set\_title(‘Lugares Rio de Janeiro’)

ax.set\_xlim(BBox[0],BBox[1]) #Acota las longitudes dentro del cuadrado BBox que habíamos definido.

ax.set\_ylim(BBox[2],BBox[3]) #Acota las latitudes dentro del cuadrado BBox que habíamos definido.

ax.scatter(df\_rio.longitud, df\_rio.latitud, zorder=1, alpha=1, c=’b’, s=30)

ax.imshow(ruh\_m, zorder=0, extent= BBox, aspect =’auto’)

for i, txt in enumerate(df\_rio.lugar):

ax.annotate(txt, (df\_rio.longitud[i], df\_rio.latitud[i]))

