

Capítulo 1

Introducción a ciencia de datos

1.1. Reducción de datos

La calidad de los datos y la cantidad de información útil que contiene son factores clave para determinar si un algoritmo *aprenderá* correctamente. Por tanto, es muy importante examinar y preprocesar nuestros conjuntos de datos antes de ingresarlos a un programa.

Una de las razones principales del gran crecimiento de la ciencia de datos, es el crecimiento acelerado de conjuntos de datos que provienen de diferentes fuentes. Por esto, no resulta simple extraer información: la mayoría de las veces los datos recolectados se obtienen sin planeación y sin previsión de un posible análisis.

Por ejemplo los *logs* de transacciones bancarias se diseñaron para propósitos de logística y contabilidad; sin embargo dentro de esos datos hay información valiosa de los hábitos y comportamientos de los consumidores. El científico de datos deberá realizar el análisis y desarrollar una estrategia *inteligente* para extraer esa información. Utilizar *mapeos* de datos ayudará en el desarrollo de dicha estrategia.

Programa: _____
Consiste en hacer un código que se encargue de cifrar y descifrar texto de acuerdo al método mostrado a continuación:

1. Este tipo de cifrado por columna con palabra clave consiste en formar una tabla con tantas columnas como letras tenga la palabra clave; a continuación, se escribe el texto en la tabla de izquierda a derecha y de arriba hacia abajo (sin espacios y, si hace falta, se rellenan los espacios de la última fila con algún carácter):

Texto: LA CRIPTOGRAFIA ES ROMANTICA
Clave: HOLLA

H	O	L	A
L	A	C	R
I	P	T	O
G	R	A	F
I	A	E	S
R	O	M	A
N	T	I	C
A	S	S	S

2. A continuación se reordenan las columnas alfabéticamente de acuerdo a la palabra clave (si hay repetición, el criterio de desempate es el orden de aparición en la palabra):

A	H	L	O
R	L	C	A
O	I	T	P
F	G	A	R
S	I	E	A
A	R	M	O
C	N	I	T
S	A	S	S

3. Finalmente se toman los caracteres por columna de arriba hacia abajo y de izquierda a derecha obteniendo finalmente el texto codificado:

ROFSACSLIGIRNACTAEMISAPRAOTS

Para descifrar un texto codificado con este método, es necesario saber la palabra clave, y a continuación se aplican las operaciones siguientes, mostradas para descifrar el ejemplo anterior:

Texto: ROFSACSLIGIRNACTAEMISAPRAOTS

Clave: HOLA

1. Se divide el texto en tantas partes como letras tiene la palabra clave (es exacta):

Grupos: ROFSACS LIGIRNA CTAEMIS APRAOTS

2. Se ordena alfabéticamente la palabra clave:

Clave: HOLA
Ordenada: AHLO

3. Se coloca cada grupo bajo cada letra de la palabra clave ordenada:

A	H	L	O
R	L	C	A
O	I	T	P
F	G	A	R
S	I	E	A
A	R	M	O
C	N	I	T
S	A	S	S

4. Se reacomoda la palabra clave junto con su columna correspondiente:

H	O	L	A
L	A	C	R
I	P	T	O
G	R	A	F
I	A	E	S
R	O	M	A
N	T	I	C
A	S	S	S

5. Se concatena cada línea de la tabla para obtener el texto el claro:

LACRIPTOGRAFIAESROMANTICASSS



*

1.1.1. Mapeo de datos y construcción de diccionarios

Desde el punto de vista matemático, un algoritmo de reducción de datos es una secuencia de mapeos de datos, es decir, funciones que consumen conjuntos de datos y producen otros conjuntos en forma reducida.

1.1.1.1. Diccionarios

Un diccionario es un conjunto desordenado de pares *clave-valor* (*keys-values*), similares a los *hashtables*, o los *arreglos asociativos* de otros lenguajes. Los diccionarios de Python están optimizados para recuperar un valor a partir de la clave, pero no al contrario.

Búsqueda por transformación de llaves

El cómputo actual y la facilidad de acceso a Internet, han hecho que grandes cantidades de información sean accesibles a prácticamente cualquier persona. La habilidad de realizar búsquedas eficientes en tales cantidades de información es fundamental para poder procesarla.

La búsqueda binaria proporciona un medio para reducir el tiempo requerido de buscar en una lista, sin embargo este método exige que los datos estén ordenados; existen otros métodos que pueden aumentar la velocidad de búsqueda en los datos y no necesitan estar ordenados.

Este método se denomina *hash* o transformación de llaves. El método *hash* consiste en convertir la clave dada numérica o alfanumérica en una dirección de memoria (índice). La correspondencia entre las claves y la dirección en el medio de almacenamiento o el arreglo se establece por una función de conversión (función *hash*). El término *hash* proviene, aparentemente, de la analogía con el significado estándar (en inglés) de dicha palabra en el mundo real: picar y mezclar.

Existen numerosos métodos de transformación de claves todos ellos tienen en común la necesidad de convertir en direcciones. En esencia la función de conversión equivale a un calculador de direcciones. Cuando se desea localizar un elemento de clave x el indicador de direcciones devolverá la posición del arreglo en la que se encuentra el elemento x .

Funciones *hash*

Idealmente, se desea obtener una estructura de datos en la que las operaciones de *insertar* y *buscar* sean $O(1)$ en el peor caso. Sin embargo, esto sólo sucede si se tiene conocimiento *a priori*, es decir, se debe saber de antemano qué elementos se almacenarán; desafortunadamente, en general no se tiene esta información. Por lo tanto, dado que en general no se puede garantizar $O(1)$ para el peor caso, será deseable obtener $O(1)$ para el caso promedio.

Para obtener el rendimiento deseado, es evidente que estas operaciones deben realizarse sin utilizar una búsqueda. Es decir, dado un elemento x , debemos determinar directamente de x la posición del arreglo en la cual se almacenará.

Se intenta diseñar un *contenedor* que será utilizado para almacenar algunos elementos de un conjunto K , en este contexto, los elementos de K se llaman *llaves* (*keys*). La estrategia general es almacenar esas llaves en un arreglo y la posición en la cual se debe almacenar cada llave en el arreglo estará dada por la función *hash* aplicada a cada llave.

Por tanto, requerimos una función $h : K \rightarrow \{0, 1, \dots, M - 1\}$. Esta función mapea el conjunto de valores a almacenar a índices dentro del arreglo de tamaño M . Ahora veremos algunas funciones *hash* comúnmente utilizadas; se asume que el conjunto de llaves es el conjunto de enteros, es decir $K = \mathbb{Z}$.

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Figure 1.1: Método de división

Método de división

Es el método más simple de función *hash* y consiste de dividir un entero x entre M y utilizar el residuo para obtener la posición del elemento. En este caso, la

función de *hash* es:

$$h(x) = |x| \bmod M$$

Generalmente esta opción es buena para casi cualquier valor de M ; sin embargo, hay situaciones en las que debe ponerse cuidado extra al elegir un valor de M apropiado. Es muy recomendable utilizar un número primo para M .

Método de la mitad del cuadrado

La operación de división es bastante lenta comparada con otro tipo de operaciones en la computadora. Si evitamos el uso de la división, se puede mejorar el tiempo de ejecución de la función *hash*. Se puede evitar el uso de la división al realizar corrimientos a nivel de bits.

Este método trabaja de la siguiente forma, primero, M debe ser una potencia de 2, es decir, $M = 2^k$ para algún valor $k \geq 1$. Para obtener la posición de algún valor entero x , se aplica la siguiente función de *hash*:

$$h(x) = \frac{W}{M}x^2 \bmod W$$

Con W igual al tamaño de la palabra de la máquina, por ejemplo, $W = 2^{32}$.

Es importante notar que, dado que M y W son potencias de 2, la proporción $W/M = 2^{w-k}$ también es una potencia de 2. Por tanto al multiplicar el término $x^2 \bmod W$ por W/M , simplemente se realiza un corrimiento a la derecha de $w - k$ bits. Es decir, extraemos k bits de la mitad (parte central) del cuadrado de la llave.

Este método es muy bueno, pero presenta problemas de colisiones cuando los números tienen muchos ceros al inicio o al final.

Método de multiplicación

Se trata de una simple variación del método de la mitad del cuadrado que sirve para evitar colisiones. En lugar de multiplicar a x por sí mismo, se multiplica por una constante a elegida cuidadosamente y extrae los k bits del centro del resultado. La función *hash* es:

$$h(x) = \frac{W}{M}ax \bmod W$$

¿Que valor de a es apropiado? Como se desea evitar el problema que presenta el método de la mitad del cuadrado con números con muchos ceros al inicio o al final, el valor elegido para a deberá cumplir la propiedad de no presentar gran cantidad de ceros al inicio o al final.

Si se elige como a un primo relativo de W , entonces existe otro número a' tal que $aa' = 1 \bmod W$. En otras palabras, a' es el inverso de $a \bmod W$, dado que el producto entre a y su inverso es 1. Tal número cumple la propiedad de que si una llave x se multiplica por a para obtener ax , se puede recuperar el valor original multiplicando por a' : $aa'x = axa' = x$.

Existen muchas posibilidades de constantes que cumplen las propiedades deseadas. Uno muy adecuado para aritmética de 32 bits es $a = 2654435769$, la representación binaria de este número no tiene muchos ceros al inicio ni al final. Además es primo relativo de $W = 2^{32}$ y el inverso de $a \bmod W = a' = 340573321$.

Método de Fibonacci

Es exactamente igual al método de multiplicación, pero utilizando un valor muy especial de a . Este valor está muy relacionado con un número llamado *proporción dorada*.

La proporción dorada se define como: dados dos enteros positivos x y y , su proporción es dorada si la proporción de x a y es la misma que la de $x + y$ a x . El valor de la proporción dorada puede determinarse como:

$$\begin{aligned}\frac{x}{y} = \frac{x+y}{x} &\Rightarrow 0 = x^2 + xy + y^2 \\ &\Rightarrow 0 = \phi^2 + \phi + 1 \\ &\Rightarrow \phi_{1,2} = \frac{1 \pm \sqrt{5}}{2}\end{aligned}$$

Recordando que el n -ésimo número de Fibonacci es de la forma:

$$f_n = \frac{1}{\sqrt{5}} (\phi_1^n - \phi_2^n)$$

El método de Fibonacci es esencialmente el método de multiplicación en el que la constante a es un entero primo relativo de W y que es el más cercano a W/ϕ . La tabla muestra valores recomendados para varios tamaños de palabra:

W	$a \approx W/\phi$
2^{16}	40503
2^{32}	2654435769
2^{64}	11400714819323198485

Características de funciones *hash*

¿Qué características son deseables para funciones hash? Una buena función *hash*:

- ◇ Evita colisiones.
- ◇ Reparte equitativamente las llaves en el arreglo.
- ◇ Es fácil de computar.

Colisiones

Idealmente, dado un conjunto de $n < M$ llaves distintas $\{k_1, k_2, \dots, k_n\}$, el conjunto de valores determinados por la función *hash* $\{h(k_1), h(k_2), \dots, h(k_n)\}$, no contiene duplicados. En la práctica, a menos que se tenga conocimiento extra sobre las llaves, no se puede garantizar la ausencia de colisiones.

Sin embargo, en ciertas condiciones se puede utilizar algún conocimiento genérico de los datos. Por ejemplo, si las llaves son números telefónicos y sabemos que los teléfonos son todos de la misma zona geográfica, no tendría sentido utilizar los códigos de área en la función hash, dado que para todos es el mismo código.

Repartición equitativa de llaves en el arreglo

Sea p_i la probabilidad de que la función hash $h(\cdot) = i$. Una función hash que reparte equitativamente llaves cumple que $p_i = 1/M$ para todo $0 < i < M$. En otras palabras, los valores devueltos por la función $h(\cdot)$ se distribuyen uniformemente. Desafortunadamente, para garantizar esto, es necesario tener conocimiento extra sobre las llaves.

Por ejemplo, el método de división presenta la desventaja de colocar llaves consecutivas de forma consecutiva en el arreglo.

En cambio, en el método de Fibonacci las llaves consecutivas se colocan entre los dos valores más separados de los ya calculados.

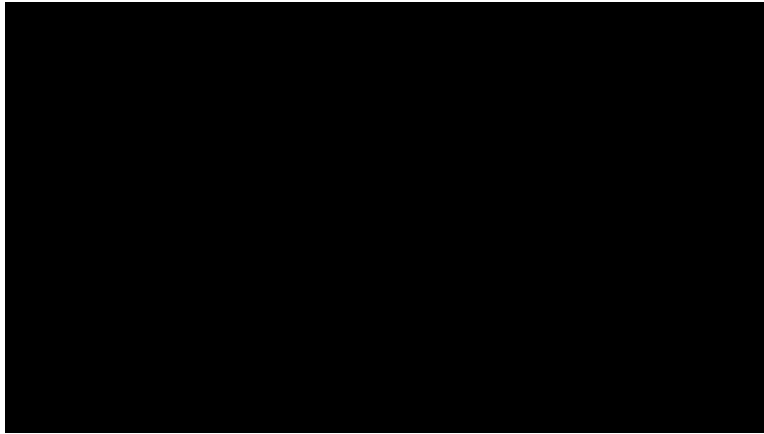


Figure 1.2: *Hash* por el método de Fibonacci

Fácilidad de cómputo

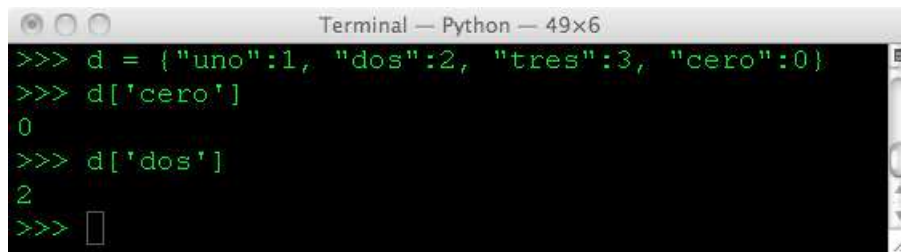
Esto no significa que sea fácil de realizar el cálculo para alguna persona y tampoco significa que sea sencillo escribir el programa que calcula la función; significa que el tiempo de ejecución debe ser $O(1)$.

Tablas de símbolos: Dicionarios

Un caso particular de los métodos de *hash*, son las llamadas *tablas de símbolos*. Su objetivo principal es asociar un *valor* con una *llave* (comúnmente una cadena). Se pueden insertar parejas clave-valor esperando que posteriormente se puede recuperar el valor asociado a una llave dada.

Estas operaciones son tan importantes en muchas aplicaciones computacionales que las tablas de símbolos se encuentran disponibles en muchos lenguajes de programación, por ejemplo:

- ◇ En php se dispone de *arreglos asociativos*.
- ◇ Python provee el tipo básico *dictionary*.
- ◇ En java se utiliza la clase *Hashtable*, etc.

A screenshot of a terminal window titled "Terminal — Python — 49x6". The terminal shows a Python dictionary `d = {"uno":1, "dos":2, "tres":3, "cerro":0}` and two lookups: `d['cerro']` returning `0` and `d['dos']` returning `2`. The prompt `>>>` is shown at the end of the last line.

```
>>> d = {"uno":1, "dos":2, "tres":3, "cerro":0}
>>> d['cerro']
0
>>> d['dos']
2
>>> 
```

Figure 1.3: Diccionario en Python

Para el correcto funcionamiento de las tablas de símbolos, se suelen imponer algunas restricciones tanto para las llaves como para los valores asociados. Algunas de las restricciones más comunes se describen a continuación.

Llaves duplicadas

Se adoptan las siguientes convenciones:

- ◇ Solo un valor es asociado a cada llave (no se permiten llaves duplicadas).
- ◇ Cuando se desea insertar una pareja llave-valor en una tabla que ya contiene la llave (y algún valor asociado), el valor nuevo reemplaza al valor anterior.

Este par de convenciones definen un arreglo asociativo, en este caso, la tabla de símbolos que puede verse como un arreglo en el que los índices son las llaves y los elementos son los valores asociados.

En un arreglo convencional, las llaves son valores enteros que sirven para acceder rápidamente a sus elementos; en un arreglo asociativo, la llaves son de otro tipo, y pueden usarse para acceder rápidamente a sus valores asociados.

Llaves nulas

Usualmente, no se acepta el valor nulo como llave (Python permite). En muchos lenguajes de programación, permitir el uso de este valor como llave resultará en una excepción en tiempo de ejecución.

Valores nulos

Al igual que con las llaves, es común evitar el uso del valor nulo como valor asociado.

1.1.1.2. Manipulando datos faltantes

Es común en aplicaciones *reales* que nuestros datos tengan uno o más valores faltantes por diversos motivos: un error en la lectura, una pregunta no respondida en una encuesta, etc. Lo que se observa en los datos es simplemente un faltante en esa ubicación (*NaN*, *None*, *Null*).

Desafortunadamente muchos algoritmos no poseen la capacidad de manipular dichos datos, incluso podrían producir resultados impredecibles si se ignoran: se deben tomar en cuenta esos valores faltantes antes de proceder con el análisis.

Consideremos el ejemplo siguiente.

```
import pandas as pd
from io import StringIO

datos = \
"""A,B,C,D
1.0,2.1,3.7,
5.9,6.2,,8.6
9.3,0.4,1.8,"""
df = pd.read_csv(StringIO(datos))
df
```

	A	B	C	D
0	1.0	2.1	3.7	4.5
1	5.9	6.2	NaN	8.6
2	9.3	0.4	1.8	NaN

En éste, los valores faltantes en los datos se reemplazan por *NaN* en el *DataFrame* de pandas.

En *dataframes* más grandes puede resultar muy tedioso buscar manualmente los valores faltantes; podemos utilizar el método *isnull* para indicar si una celda contiene un valor numérico y con *sum* se obtiene el número de valores faltantes por columna.

```
df.isnull().sum
```

```
A      0
B      0
C      1
D      1
dtype: int64
```

Eliminando valores faltantes

Una de las maneras más simples de manipular los valores faltantes es eliminar características (columnas, *features*) o muestras (filas) que contengan valores faltantes.

Para eliminar filas:

```
df.dropna(axis=0)
```

	A	B	C	D
0	1.0	2.1	3.7	4.5

Para las columnas:

```
df.dropna(axis=1)
```

	A	B
0	1.0	2.1
1	5.9	6.2
2	9.3	0.4

Existen más parámetros para este método, el siguiente sirve para indicar que elimine aquellas muestras en las que todas sus columnas sean *NaN*:

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

	A	B	C	D
0	1.0	2.1	3.7	4.5
1	5.9	6.2	NaN	8.6
2	9.3	0.4	1.8	NaN

Eliminar muestras que tengan menos de cuatro valores:

```
df.dropna(thresh=4)
```

	A	B	C	D
0	1.0	2.1	3.7	4.5

Eliminar muestras que falten valores en columnas en particular:

```
df.dropna(subset=['C'])
```

```
df.dropna(subset=['A','C'])
```

	A	B	C	D
0	1.0	2.1	3.7	4.5
2	9.3	0.4	1.8	NaN

Sustituyendo valores faltantes

En ocasiones no es recomendable eliminar características o muestras completas debido a que se podrían perder datos valiosos en el proceso; en este caso se pueden utilizar técnicas de interpolación para estimar los valores faltantes a partir de otras muestras en nuestro conjunto de datos. Una usada comúnmente es la media por características (columnas).

Por ejemplo:

```
datos_n = df.copy() # No afectar al df original
for col in df.columns.values:
    falta = np.sum(df[col].isnull())
    if falta:
        print('Asignando {} valores en columna : {}'.format(falta,col))
        mean = df[col].mean()
        datos_n[col] = df[col].fillna(mean)
datos_n
```

Se puede utilizar la clase *SimpleImputer* para sustituir los datos faltantes:

```
from sklearn.impute import SimpleImputer

imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit(df.values)
datos_n = imp.transform(df.values)
datos_n
```

```
array([[1.   , 2.1  , 3.7  , 4.5  ],
       [5.9  , 6.2  , 2.75, 8.6  ],
       [9.3  , 0.4  , 1.8  , 6.55]])
```

Nota: En versiones anteriores de *sklearn*, se utilizaba:

```
from sklearn.preprocessing import Imputer
```

Otras opciones para el parámetro *strategy* son *median* y *most_frequent*, este último coloca el valor más repetido en lugar de los faltantes y es muy útil cuando se tienen datos categóricos:

```
import pandas as pd
from sklearn.impute import SimpleImputer

df = pd.DataFrame([["a", np.nan],
                    ["a", "y"],
                    ["b", "y"],
                    ["c", "x"],
                    [np.nan, "z"]], dtype="category")
imp = SimpleImputer(strategy="most_frequent")
print(imp.fit_transform(df))
```

```
[['a' 'y']
 ['a' 'x']
 ['b' 'y']
 ['c' 'y']
 ['a' 'z']]
```

Data Cleaning in Python: the Ultimate Guide (<https://bit.ly/2WAinjc>).

1.1.1.3. Manipulando datos categóricos

Comúnmente los conjuntos de datos contienen datos categóricos. Es importante distinguir entre datos categóricos **ordinales** y **nominales**; los ordinales son aquellos que pueden ser ordenados (p.e., en la talla de una prenda $M < G < XG$); en los nominales no tiene mucho sentido pensar en un orden (p.e., en el color de una prenda).

```
import pandas as pd

df = pd.DataFrame([
    ['verde', 'M', 10.3, 'clase1'],
    ['rojo', 'G', 14.2, 'clase2'],
    ['azul', 'XG', 15.6, 'clase1'] ])
df.columns = ['color', 'talla', 'precio', 'clase']
df
```

	color	talla	precio	clase
0	verde	M	10.3	clase1
1	rojo	G	14.2	clase2
2	azul	XG	15.6	clase1

Este conjunto de datos contiene una característica *nominal* (color) una *ordinal* (talla) y una *numérica* (precio). Este conjunto puede utilizarse en un algoritmo supervisado, dado que las etiquetas de la clase a la que pertenece cada muestra en la última columna.

Mapeando valores ordinales

Para garantizar que nuestros algoritmos interpretan correctamente características ordinales, es necesario convertir estas características en valores enteros; desafortunadamente no hay una función que pueda determinar automáticamente el orden correcto a partir de las etiquetas de la talla; por lo tanto, en general se debe realizar este proceso manualmente (con ayuda de un diccionario):

```
talla_map = {'XG':3, 'G':2, 'M':1}
df['talla'] = df['talla'].map(talla_map)
df
```

	color	talla	precio	clase
0	verde	1	10.3	clase1
1	rojo	2	14.2	clase2
2	azul	3	15.6	clase1

Codificando las etiquetas de clase

Muchos algoritmos de aprendizaje requieren que las etiquetas de clase estén codificadas como números enteros. Aunque la mayoría de los algoritmos que provee *scikit-learn* pueden realizar esta conversión internamente, es una buena práctica proveer las etiquetas como enteros y como las clases no son ordinales no importa qué número se asigna a cada una de ellas. Por lo tanto, podemos enumerar las etiquetas de clase comenzando con 0:

```
import numpy as np

class_map = {et:idx for idx,et in
              enumerate(np.unique(df.clase))}
class_map
```

```
{'clase1': 0, 'clase2': 1}
```

A continuación podemos usar el diccionario para realizar la conversión de las etiquetas de clase a enteros:

```
df['clase'] = df['clase'].map(class_map)
df
```

	color	talla	precio	clase
0	verde	1	10.3	0
1	rojo	2	14.2	1
2	azul	3	15.6	0

Convenientemente existe la clase *LabelEncoder* implementada *scikit-learn*.

Codificando en formato *one-hot* valores nominales

Podemos realizar un proceso similar para la columna *color* (dado que no se requiere orden).

```
import numpy as np

color_map = [[c,idx] for idx,c in enumerate(np.unique(df.color))]
color_map
```

```
[['azul', 0], ['rojo', 1], ['verde', 2]]
```

Si nos detuviéramos en este punto, estaríamos cayendo en uno de los errores más comunes cuando se utilizan datos categóricos: los colores no tienen un orden particular; sin embargo un algoritmo de aprendizaje supondría que verde *es mayor que* rojo y que rojo *es mayor que* azul. Esta suposición no es correcta y aunque se podría obtener un buen resultado, en general puede no ser óptimo.

Una posible solución es utilizar la técnica conocida como **codificación *one-hot***. La idea básica es crear una nueva característica *ficticia (dummy)* por cada valor único de la columna nominal correspondiente. En este caso, se generarán tres columnas: *azul*, *rojo* y *verde*. Se utilizan valores binarios para indicar el color particular de cada muestra.

Para realizar esta transformación se puede utilizar el *OneHotEncoder* disponible en el módulo *preprocessing* de *sci-kitlearn*:

```
from sklearn.preprocessing import OneHotEncoder
# Ignorar las categorías no usadas en el ajuste (fit)
ohe = OneHotEncoder(handle_unknown='ignore')
ohe.fit(color_map)
ohe.transform(color_map).toarray()

array([[1., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 1.]])
```

¿No deberían ser solamente tres columnas?

```
ohe.get_feature_names(['color', 'grupo'])

array(['color_azul', 'color_rojo', 'color_verde',
       'grupo_0', 'grupo_1', 'grupo_2'], dtype=object)
```

Una forma más conveniente para crear esas características ficticias para la codificación *one-hot*, es utilizar el método `get_dummies` implementado en `pandas`. Si se aplica a un `DataFrame`, este método convertirá solamente las columnas de tipo `string` y dejará las otras columnas sin cambios:

```
pd.get_dummies(df[['precio', 'color', 'talla']])
```

	precio	talla	color_azul	color_rojo	color_verde
0	10.3	1	0	0	1
1	14.2	2	0	1	0
2	15.6	3	1	0	0

Es importante tomar en cuenta que al utilizar este tipo de codificación se pueden obtener problemas para algunos métodos (p.e., métodos que requieren calcular la matriz inversa). Si las características tienen correlación alta, las matrices son muy difíciles de invertir.

Para reducir la correlación de las variables, se puede simplemente eliminar una característica del arreglo *one-hot* obtenido, es importante notar que no se pierde información al eliminar una columna. El parámetro `drop_first` de `get_dummies` puede ser de ayuda:

```
pd.get_dummies(df[['precio', 'color', 'talla']],
               drop_first=True)
```

	precio	talla	color_rojo	color_verde
0	10.3	1	0	1
1	14.2	2	1	0
2	15.6	3	0	0

1.1.1.4. Escalamiento de características

Este es otro paso clave en el preprocesamiento de datos; muchos algoritmos de aprendizaje y optimización tienen mejor comportamiento si las características discriminantes se encuentran en la misma escala.

Existen dos enfoques comunes para escalar características:

1. **Normalización**, se refiere al reescalamiento de cada característica en el intervalo $[0, 1]$. Para esto, se aplica el *escalamiento min-max* en cada columna:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

Donde $x^{(i)}$ es una muestra particular, x_{min} es el valor mínimo de la columna y x_{max} el máximo.

2. **Estandarización**, este preprocesamiento puede resultar más práctico para algoritmos de optimización (p.e. *gradient descent*), esto se debe a que muchos modelos inicializan sus pesos a

valores cercanos a 0. Utilizando estandarización, se centran las columnas de características a *media* 0 con *desviación estándar* 1. Además, la estandarización mantiene información útil sobre secciones aisladas y hace que los algoritmos sean menos sensibles a ellas en contraste al escalamiento min-max.

El procedimiento de estandarización se realiza con:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Donde μ_x es la media muestral de una columna particular y σ_x es su correspondiente desviación estándar.

Por ejemplo:

```
ex = np.array([0,1,2,3,4,5])
print('Normalizado      : ', (ex-ex.min()/(ex.max()-ex.min())))
print('Estandarizado   : ', (ex-ex.mean())/ex.std())
```

```
Normalizado      :  [0.  0.2 0.4 0.6 0.8 1. ]
Estandarizado   :  [-1.46385011 -0.87831007 -0.29277002  0.29277002
 0.87831007  1.46385011]
```

Ambas formas de escalamiento están disponibles en *scikit-learn* y se utilizan de la siguiente manera:

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train) # datos de entrenamiento
X_test_norm = mms.transform(X_test) # datos de prueba

from sklearn.preprocessing import StandardScaler
stds = StandardScaler()
X_train_std = stds.fit_transform(X_train) # datos de entrenamiento
X_test_std = stds.transform(X_test) # datos de prueba
```

1.1.1.5. Seleccionando características significativas

Si se observa que un modelo tiene mejor rendimiento con el conjunto de entrenamiento que con el de pruebas, es un indicador de *sobreajuste*; esto significa que el modelo adecúa sus parámetros a las observaciones de los datos de entrenamiento, pero no generaliza bien a nuevos datos.

Entre las posibles soluciones al sobreajuste, se tiene la reducción de la dimensión de los datos.

Selección secuencial de características

Los algoritmos de selección secuencial son una familia de algoritmos voraces (*greedy*) usados para reducir la dimensión d de características a un subespacio de dimensión k que cumple $k <$

d. El objetivo es seleccionar automáticamente un subconjunto de características que sean las más relevantes para el problema; con esto se puede mejorar la eficiencia o reducir errores de generalización del modelo al eliminar características irrelevantes o ruido.

Un algoritmo de selección común es el llamado ***Sequential Backward Selection (SBS)*** cuyo objetivo es reducir la dimensión del espacio de características con un mínimo de pérdidas en el rendimiento del modelo mejorando la eficiencia computacional. Además, SBS puede mejorar también el poder predictivo de un modelo si está sobreajustado.

La idea detrás de SBS es simple: eliminar secuencialmente características de un conjunto hasta que se tenga el número deseado de ellas. Para determinar qué columna eliminar se debe establecer una función criterio \mathcal{J} ; uno muy usado es simplemente comparar el rendimiento del modelo antes y después de eliminar una columna; la característica a eliminar será aquella que cause el menor descenso en el rendimiento. El algoritmo puede describirse en cuatro pasos:

Algoritmo 1.1 *Sequential Backward Selection*

1. Inicializar el algoritmo con $k = d$, d es la dimensión del espacio original de características \mathbf{X}_d
 2. Determinar la característica x^- que maximiza el criterio: $x^- = \arg \max (\mathcal{J}(X_k - x))$, donde $x \in X_k$
 3. Eliminar la característica x^- del conjunto: $X_{k-1} = X_k - x^-$; $k = k - 1$
 4. Si k es mayor que el número deseado de características, repetir desde el paso 2; de otro modo, terminar la ejecución
-

Reducción de dimensión no supervisada: Análisis de Componentes Principales (PCA)

Otra forma de reducir la cantidad de datos a procesar, es la compresión de datos, dado que nos permite almacenar y analizar gran cantidad de datos producidos y recolectados por medio tecnológicos. Mientras que SBS es un algoritmo de *selección*, PCA (***Principal Component Analysis***) es un algoritmo de *extracción* de características. La diferencia principal es que en un algoritmo de selección, se conservan las características originales y en uno de extracción, los datos se transforman o proyectan en un nuevo espacio de características.

PCA ayuda a identificar patrones en los datos basado en la correlación entre las características; es decir, intenta encontrar las direcciones de máxima varianza en datos con muchas dimensiones y las proyecta en un nuevo subespacio con igual o menos dimensiones que el original. El algoritmo se compone de los siguientes pasos:

Algoritmo 1.2 *Principal Component Analysis*

1. Estandarizar el conjunto de datos de dimensión d
 2. Obtener la matriz de covarianza
 3. Descomponer la matriz de covarianza en sus eigenvalores y eigenvectores
 4. Ordenar los eigenvalores de manera decreciente de acuerdo a sus correspondientes eigenvectores
 5. Seleccionar los k eigenvectores que corresponden con los k mayores eigenvalores; k es la dimensión de nuevo subespacio de características ($k < d$)
 6. Construir una matriz de proyección \mathbf{W} con los primeros k eigenvectores
 7. Transformar el conjunto de datos de entrada \mathbf{X} de dimensión d utilizando la matriz de proyección \mathbf{W} para obtener el nuevo subespacio de características de dimensión k
-

Implementación de PCA desde cero:

```
# Datos
import pandas as pd
df_wine = pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data',
    header=None)

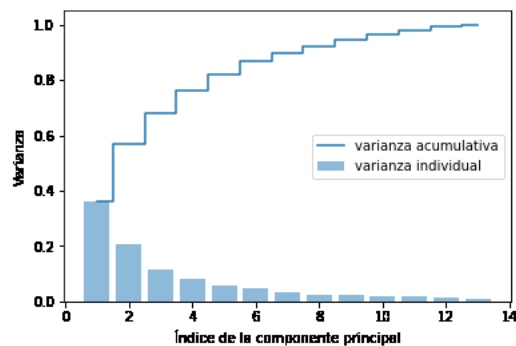
# Separar datos de entrenamiento y prueba
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:,1:].values, df_wine.iloc[:,0:].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
# estadarizamos
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# Matriz de covarianza eigenvalores & eigenvectores
import numpy as np
cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
print('Eigenvalores : \n', eigen_vals)
```

```

# Gráfica que muestra los aportes de cada eigenvalor
# y la suma acumulativa
import matplotlib.pyplot as plt
tot = sum(eigen_vals)
var_exp = [(ev/tot) for ev in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
plt.bar(range(1,14),var_exp,alpha=0.5,align='center',label='varianza individual')
plt.step(range(1,14),cum_var_exp,where='mid',label='varianza acumulativa')
plt.xlabel('Índice de la componente principal')
plt.ylabel('Varianza')
plt.legend(loc='best')
plt.show()

```



```

# Transformación de características
# lista de tuplas (e_val, e_vec)
eigen_par = [(np.abs(eigen_vals[i]),eigen_vecs[:,i])
              for i in range(len(eigen_vals))]
# ordenar de mayor a menor e_val
eigen_par.sort(key=lambda k: k[0], reverse=True)

```

```

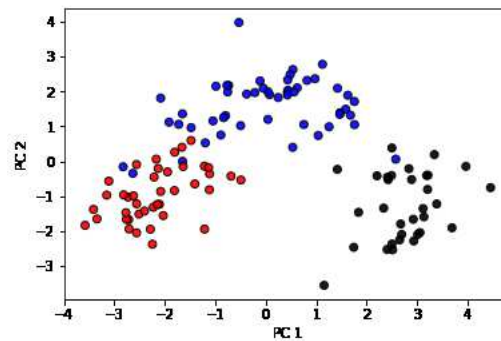
# Seleccionar 2 características con los eigenvalores más grandes (~60%) para
# graficarlos, en la 'realidad' se debe analizar cuánto desea mantenerse del 1
w = np.hstack((eigen_par[0][1][:,np.newaxis],
               eigen_par[1][1][:,np.newaxis]))
# Matriz de proyección W de 13 x 2, a partir de los 2 eigenvectores principales
print('Matriz W : \n', w)

```

```
# Con ayuda de la matriz de proyección se puede transformar
# una muestra (1x13) al subespacio de PCA (1x2)
print('Original    : ', X_train_std[0])
print('Proyectado   : ', X_train_std[0].dot(w))
```

```
# Todo el conjunto de entrenamiento (124x13 a 124x2)
X_train_pca = X_train_std.dot(w)
```

```
# Visualización del conjunto wine transformado como un scatter de 124x2
colors = np.array(['lime', 'red', 'blue', 'black', 'lightgreen', 'cyan'])
y = y_train[:,0]
plt.scatter(X_train_pca[:,0],
            X_train_pca[:,1],
            alpha=0.9,
            c=colors[y.astype(int)],
            edgecolor='black')
```



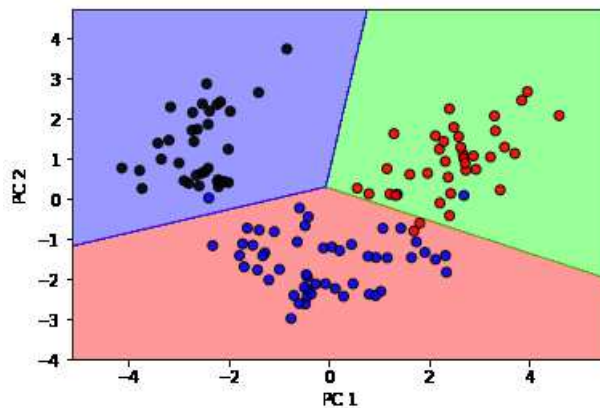
Usando PCA de *scikit-learn*:

```
# Bibliotecas
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
# Datos
df_wine = pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data',
    header=None)
```

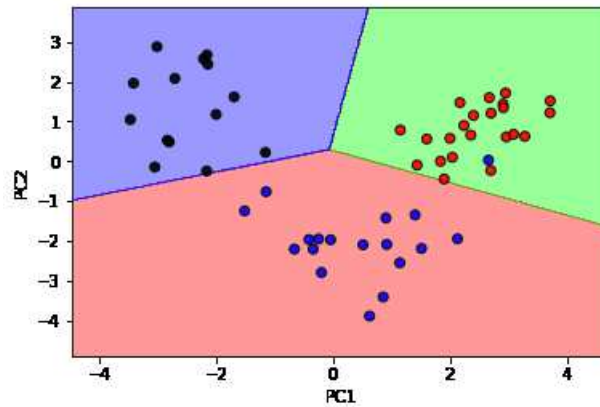
```
# Separar datos de entrenamiento y prueba
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:,1:].values, df_wine.iloc[:,0:].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
# estandarizamos
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

```
# Probaremos con una regresión logística
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
# inicializar PCA y el modelo de RL
pca = PCA(n_components=2)
lr = LogisticRegression(multi_class='auto', solver='liblinear')
# Ajustar y transformar los datos
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
lr.fit(X_train_pca, y_train[:,0])
```

```
# Graficar el conjunto de entrenamiento
plot_decision_regions(X_train_pca, y_train[:,0], classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.show()
```



```
# Graficar los resultados para el conjunto de prueba
plot_decision_regions(X_test_pca, y_test[:,0], classifier=lr)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```



Programa: _____

- ◇ Implementar el algoritmo SBS desde cero
- ◇ Implementar el cálculo de la matriz de covarianzas

Recordar: $X = \left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 3 \\ 2 \end{pmatrix} \right\}$ $\vec{\mu} = \begin{pmatrix} 2.2 \\ 2 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0.56 & -0.2 \\ -0.2 & 0.4 \end{pmatrix}$

*

Reducción de dimensión supervisada: Análisis Lineal Discriminante (LDA)

Linear Discriminant Analysis (LDA) es una técnica de extracción de características que puede usarse para incrementar la eficiencia computacional y reducir los sobreajustes. Formulado inicialmente por Ronald A. Fischer (<https://bit.ly/3f9e6KF>) en 1936 con el conjunto de datos de flores iris para problemas de clasificación de dos clases. En 1948 C. Radhakrishna Rao (<https://bit.ly/2VX3Hub>) lo generalizó para problemas multiclase bajo el supuesto de covarianzas de clase iguales y clases con distribuciones normales.

En general, los conceptos detrás de LDA son muy similares a PCA: mientras PCA busca las componentes ortogonales de varianza mínima, el objetivo de LDA es encontrar un subespacio de características que optimice la separabilidad de clases.

El algoritmo se compone de los siguientes pasos:

Algoritmo 1.3 *Linear Discriminant Analysis*

1. Estandarizar el conjunto de datos de dimensión d
 2. Para cada clase, calcular su vector de medias (de dimensión d)
 3. Obtener la matriz de dispersión (*scatter matrix*) entre clases S_B y la matriz de dispersión de la propia clase S_W
 4. Determinar los eigenvalores y eigenvectores correspondientes a la matriz $S_W^{-1}S_B$
 5. Ordenar los eigenvalores de manera decreciente de acuerdo a sus correspondientes eigenvectores
 6. Seleccionar los k eigenvectores que corresponden con los k mayores eigenvalores para construir una $d \times k$ —dimensional matriz de proyección \mathbf{W} ; los eigenvectores son las columnas de dicha matriz
 7. Proyectar las muestras sobre el nuevo subespacio de características utilizando la matriz de proyección \mathbf{W}
-

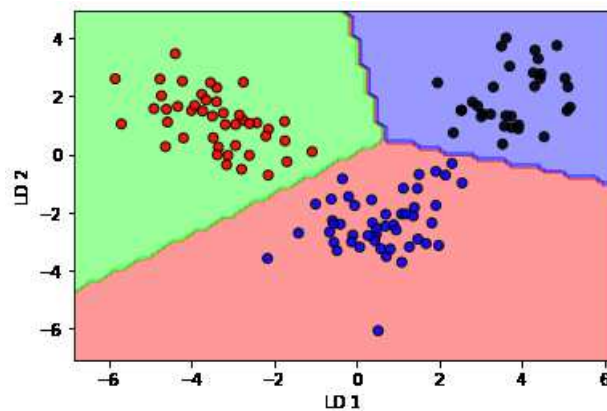
Usando LDA de *scikit-learn*:

```
# Bibliotecas
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
# Datos
df_wine = pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data',
    header=None)
```

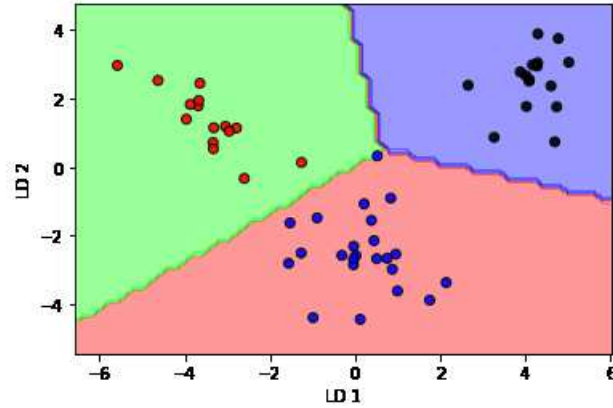
```
# Separar datos de entrenamiento y prueba
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:,1:].values, df_wine.iloc[:,0:].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
# estandarizamos
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

```
# Revisar con regresión logística
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
# inicializar LDA y el modelo de RL
lda = LDA(n_components=2)
lr = LogisticRegression()
# Ajustar y transformar los datos
X_train_lda = lda.fit_transform(X_train_std, y_train[:,0])
lr.fit(X_train_lda,y_train[:,0])
```

```
# Grafica del conjunto de entrenamiento
from mlxtend.plotting import plot_decision_regions
plot_decision_regions(X_train_lda, y_train[:,0], clf=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.show()
```



```
# Grafica del conjunto de pruebas
X_test_lda = lda.transform(X_test_std)
plot_decision_regions(X_test_lda, y_test[:,0], clf=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.show()
```



Tarea

Escribir un código que obtenga la matriz de dispersión de acuerdo a:

$$S = \sum_{j=1}^n (\mathbf{x}_j - \bar{\mathbf{x}}) (\mathbf{x}_j - \bar{\mathbf{x}})^T$$

donde $\bar{\mathbf{x}}$ es el vector de medias.

*

Reducción no lineal de dimensión: Análisis de la Componente Principal por Núcleos (KPCA)

Algunos algoritmos de aprendizaje automático suponen separabilidad lineal de los datos de entrada; por ejemplo, el perceptrón requiere separabilidad perfecta para garantizar convergencia. Por otro lado, existen algoritmos que suponen que la falta de separabilidad lineal se debe a ruido; por ejemplo, la regresión logística.

Sin embargo, si tenemos un problema no lineal, muy comunes en aplicaciones reales, las técnicas de transformación lineal para reducir la dimensión (PCA, LDA) pueden no ser una buena opción. En esta sección se presenta una versión *kernelizada* (por núcleos) de PCA: **Kernel Principal Component Analysis (KPCA)** para transformar datos que no son linealmente separables en un subespacio nuevo de menor dimensión que sea adecuado para clasificadores lineales.

Funciones de núcleos

Cuando tenemos problemas no lineales, se pueden abordar proyectándolos en un espacio de características de dimensión superior en el que las clases son linealmente separables. Para transformar las muestras $\mathbf{x} \in \mathbb{R}^d$ sobre un espacio superior k -dimensional, debe definirse una función de mapeo ϕ :

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k; (k \gg d)$$

ϕ puede verse como una función que genera combinaciones no lineales de las características originales del conjunto de datos d -dimensional original sobre un espacio de características k -dimensional. Por ejemplo, para un vector de dos dimensiones $\mathbf{x} \in \mathbb{R}^2$ un mapeo potencial al espacio de 3 dimensiones \mathbb{R}^3 puede ser:

$$\begin{array}{c}
\mathbf{x} = [x_1, x_2]^T \\
\downarrow \\
\phi \\
\downarrow \\
\mathbf{z} = [x_1^2, \sqrt{2x_1x_2}, x_2^2]^T
\end{array}$$

De esta forma, se puede utilizar PCA en un espacio de orden superior para después proyectarlo sobre un espacio de dimensión menor en el que las muestras puedan separarse con un clasificador lineal.

Para implementar un KPCA con *función de base radial* (**Radial Basis Function RBF** o Gaussiano) como kernel, se realizan los siguientes pasos:

Algoritmo 1.4 Kernel PCA

1. Obtener la matriz de núcleos \mathbf{K} , donde debe calcularse: $\mathbf{k}(x^{(i)}, x^{(j)}) = \exp\left(-\gamma \|x^{(i)} - x^{(j)}\|^2\right)$; $\gamma = \frac{1}{2\sigma^2}$ para cada par de muestras:

$$\mathbf{K} = \begin{bmatrix} \mathbf{k}(x^{(1)}, x^{(1)}) & \mathbf{k}(x^{(1)}, x^{(2)}) & \cdots & \mathbf{k}(x^{(1)}, x^{(n)}) \\ \mathbf{k}(x^{(2)}, x^{(1)}) & \mathbf{k}(x^{(2)}, x^{(2)}) & \cdots & \mathbf{k}(x^{(2)}, x^{(n)}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{k}(x^{(n)}, x^{(1)}) & \mathbf{k}(x^{(n)}, x^{(2)}) & \cdots & \mathbf{k}(x^{(n)}, x^{(n)}) \end{bmatrix}$$

Es decir, si tenemos 100 muestras de entrenamiento, su matriz de núcleos sería de dimensión 100×100

2. Centrar la matriz \mathbf{K} utilizando: $\mathbf{K}' = \mathbf{K} - \mathbf{1}_n \mathbf{K} - \mathbf{K} \mathbf{1}_n + \mathbf{1}_n \mathbf{K} \mathbf{1}_n$; donde $\mathbf{1}_n$ es una matriz $n \times n$ -dimensional en la que todos los valores son $\frac{1}{n}$
 3. Seleccionar los k eigenvectores de la matriz centrada que corresponden con los k mayores eigenvalores
-

El centrado de la matriz (paso 2) se requiere porque no es posible garantizar que el nuevo espacio esté centrado en cero. Otras funciones de núcleo comúnmente usadas son el *kernel polinomial* y el *kernel tangente hiperbólico* (sigmoide).

Implementación de RBF Kernel PCA:

```

# Bibliotecas
from scipy.spatial.distance import pdist, squareform
from scipy.linalg import eigh
import numpy as np

```

```

def rbf_kpca(X, gamma, n_components):
    # calcula las distancias cuadradas de todas las parejas
    # en el conjunto de datos MxN dimensional
    sq_dists = pdist(X, 'sqeuclidean')

    mat_sq_dists = squareform(sq_dists) # convertir en una matriz

    K = np.exp(-gamma * mat_sq_dists) # Obtener la matriz de núcleos

    # Centrar la matriz de núcleos
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtener los eigenpares de la matriz de núcleos centrada
    # scipy.linalg.eigh los devuelve en orden ascendente
    eigenvals, eigenvecs = eigh(K)
    eigenvals, eigenvecs = eigenvals[::-1], eigenvecs[:,::-1] # invertir su orde

    # Seleccionar los k primeros eigenvectores (muestras proyectadas)
    alphas = np.column_stack([eigenvecs[:, i] for i in range(n_components)])
    # Seleccionar los correspondientes eigenvalores
    lambdas = np.column_stack([eigenvals[:, i] for i in range(n_components)])

    return alphas, lambdas

```

Ejemplo: Separando medias lunas

Probando el código con conjuntos de datos no lineales; primero creando un *dataset* bidimensional de 100 muestras con forma de media luna:

```

# Bibliotecas
import numpy as np
import matplotlib.pyplot as plt

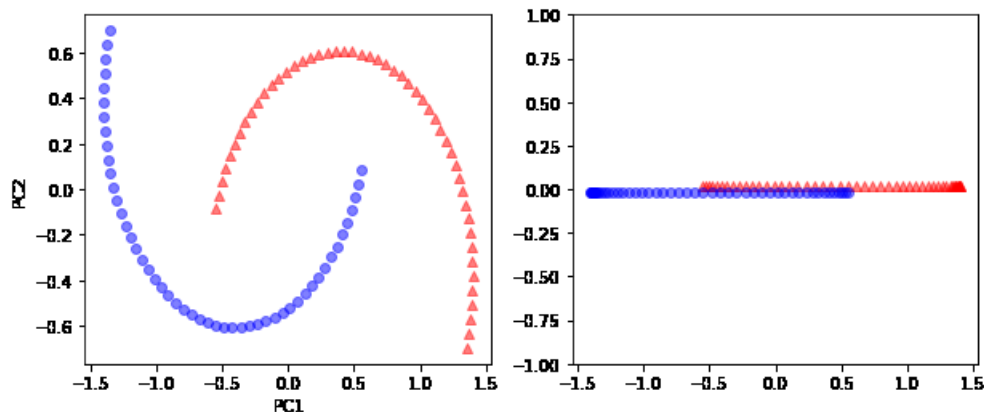
```

```
# Datos: medias lunas
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, random_state=123)
plt.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.show()
```

Primero tratemos de separar los conjuntos usando PCA:

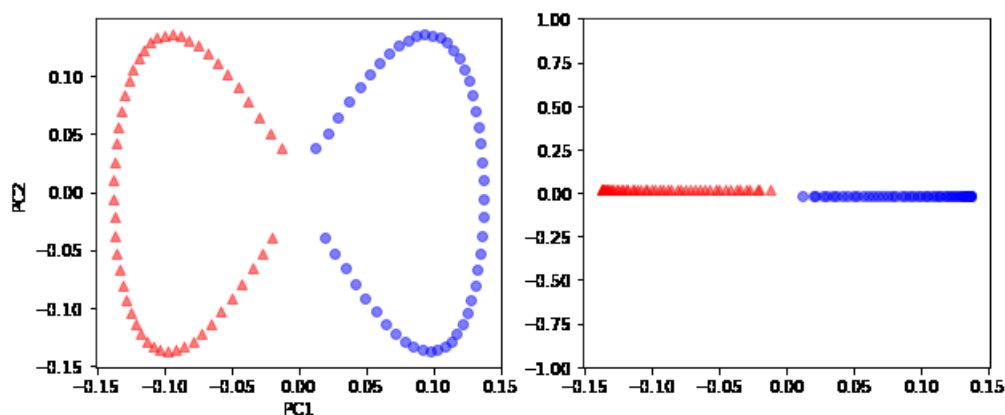
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,4))
ax[0].scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red',
              marker='^', alpha=0.5)
ax[0].scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue',
              marker='o', alpha=0.5)
ax[1].scatter(X_pca[y==0, 0], np.zeros((50,1))+0.02, color='red',
              marker='^', alpha=0.5)
ax[1].scatter(X_pca[y==1, 0], np.zeros((50,1))-0.02, color='blue',
              marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1,1])
ax[0].set_xlabel('PC1')
plt.show()
```



Es claro que PCA no puede separar los conjuntos. Intentemos con *rbf_kpca*:

```
# Separar con rbf_kpca
X_kpca, l = rbf_kpca(X, gamma=15, n_components=2)
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,4))
ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1], color='red',
              marker='^', alpha=0.5)
ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1], color='blue',
              marker='o', alpha=0.5)
ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02, color='red',
              marker='^', alpha=0.5)
ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02, color='blue',
              marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1,1])
ax[0].set_xlabel('PC1')
plt.show()
```



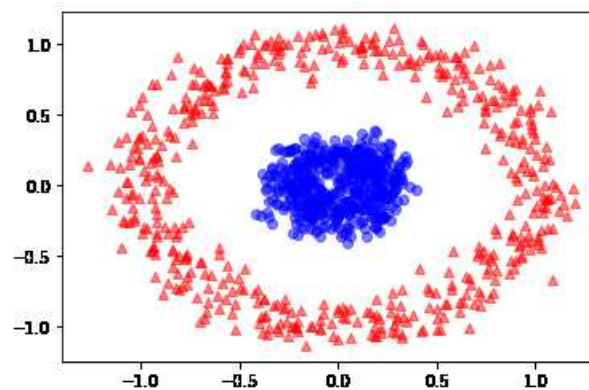
Transformó los datos a un conjunto linealmente separable. Por supuesto que existe *KernelPCA* implementado en *sklearn*:

```
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X)
# se puede graficar con el mismo código del anterior
```

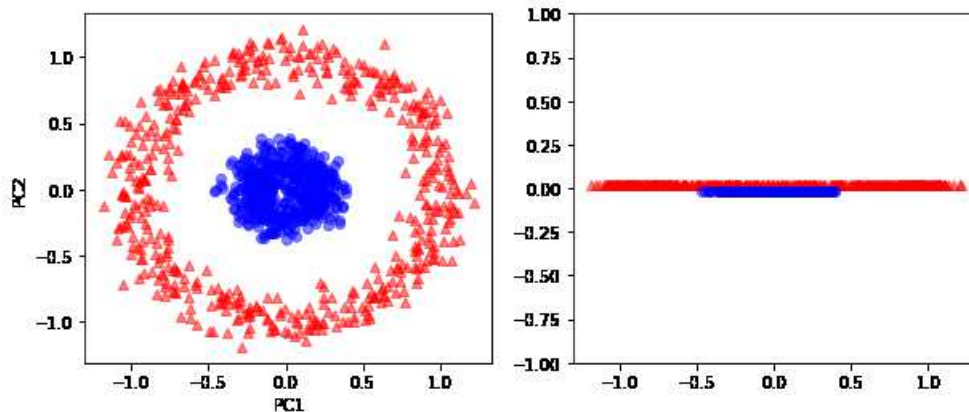
Tarea

Otro conjunto interesante para probar *rbf_kpca* son círculos concéntricos; *sklearn* incluye un generador de dicho conjunto:

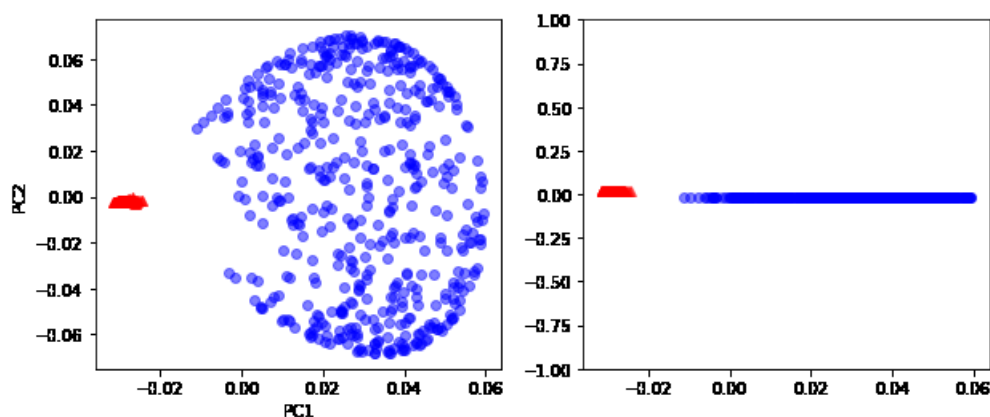
```
# Datos: círculos concéntricos
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=1000, random_state=123, noise=0.1, factor=0.2)
```



1. Probar que PCA no puede separar este conjunto:



2. Probar que tanto *rbf_kpca* (propio) como KernelPCA de *sklearn* separan el conjunto de forma correcta:



*

1.1.2. Algoritmos escalables y estadística asociativa

Es común que existan problemas que contengan cantidades de datos tan grandes que una sola computadora no pueda manipularlos. Una posible solución es reemplazar una computadora por un conjunto (*red*) de procesadores (*nodos*) en los que los datos se distribuyen y procesan. Sin embargo, la solución completa va más allá de la simple sustitución de *hardware*: es necesario contar con algoritmos que se adapten al ambiente distribuido, es decir, algoritmos escalables. La escalabilidad depende de la estadística que el algoritmo está calculando y la estadística que permite la escalabilidad es conocida como *estadística asociativa*.

El principal interés cae sobre el componente algorítmico y en particular la situación en la que cada nodo ejecuta el mismo programa sobre un subconjunto de datos. Una vez que todos los nodos han terminado sus cálculos, los resultados de todos los nodos se combinan para obtener la solución global. En general esta estrategia es buena, dado que sólo se necesita el mismo algoritmo para todos los nodos.

El resultado final no debe depender en la forma en la que se dividen los subconjuntos. De forma precisa, una partición de un conjunto A es una colección de subconjuntos A_1, \dots, A_n tal que $A = \cup_i A_i$ y $\cap_i A_i = \emptyset$. Decimos que un algoritmo es escalable si los resultados son los mismos para todas las posibles particiones de los datos; si se cumple la condición de escalabilidad, entonces *solo* se deberá incrementar el número de nodos y subconjuntos si aumenta la cantidad de datos a procesar. Por otro lado, si el algoritmo entrega soluciones diferentes dependiendo de la partición, entonces debemos determinar aquella que obtenga la mejor solución. Aunque no es claro el criterio para determinar cual es la *mejor solución*, la intuición indica que es aquella que se obtiene en una ejecución utilizando todos los datos disponibles; bajo esta premisa se consideran también escalables.

Se utiliza el término *escalable* porque la escala (tamaño) de los datos no limita el funcionamiento del algoritmo en cuestión. Por ejemplo los algoritmos de reducción de datos basados en mapeos son escalables. Para tener algoritmos escalables más sofisticados, debemos tener la capacidad de calcular otras características estadísticas, por ejemplo, el estimador de mínimos cuadrados de un vector de parámetros β o una matriz de correlación.

No toda la estadística puede calcularse con algoritmos escalables, aquella que se puede computar con este tipo de algoritmos se llama *asociativa*. La característica que define a la estadística

asociativa es que cuando el conjunto de datos se particiona en conjuntos disjuntos y se obtiene alguna característica estadística sobre cada uno de ellos, se pueden combinar para obtener el mismo resultado que si se hubiera obtenido sobre el conjunto de datos completo: si la función de un algoritmo es calcular estadística asociativa, entonces es escalable.

1.1.2.1. Estadística asociativa

Sea $D = \{x_1, x_2, \dots, x_n\}$ un conjunto de n observaciones. La i -ésima observación $x_i = [x_{i,1}, x_{i,2}, \dots, x_{i,p}]^T$ es un vector de p números reales. Una *partición* de D es una colección de suconjuntos disjuntos D_1, D_2, \dots, D_r tal que $D = D_1 \cup D_2 \dots \cup D_r$. Entonces:

$$\mathbf{s}(D) = [s_1(D), s_2(D), \dots, s_d(D)]$$

denota una **estadística asociativa** (un vector) de dimensión $d \geq 1$. Una estadística es asociativa si posee la propiedad de asociatividad y es de *baja dimensión*; este último término informal describe una estadística que puede almacenarse sin necesidad de *muchos* recursos computacionales. En la práctica, la asociatividad implica que los datos pueden particionarse en r subconjuntos y la estadística puede aplicarse a cada uno de ellos; al final de los r cálculos, se pueden utilizar para obtener el valor de la estadística aplicada al conjunto de datos completo; por tanto, no hay pérdida de información o indeterminación resultante del procesamiento distribuido y el algoritmo resultante es escalable.

Un ejemplo de estadística asociativa sobre un conjunto de n números reales $D = \{x_1, x_2, \dots, x_n\}$ es:

$$s(D) = \begin{bmatrix} \sum_{i=1}^n x_i \\ n \end{bmatrix}_{2 \times 1}$$

La asociatividad se cumple porque para cualquier partición, la estadística total puede obtenerse a partir de las estadísticas de cada partición. Por ejemplo, supóngase que D se particiona como $D_1 = \{x_1, x_2, \dots, x_m\}$ y $D_2 = \{x_{m+1}, x_{m+2}, \dots, x_n\}$; entonces:

$$\begin{aligned} \mathbf{s}(D_1) + \mathbf{s}(D_2) &= \begin{bmatrix} \sum_{i=1}^m x_i \\ m \end{bmatrix} + \begin{bmatrix} \sum_{i=m+1}^n x_i \\ n - m \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^m x_i + \sum_{i=m+1}^n x_i \\ m + n - m \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^n x_i \\ n \end{bmatrix} = \mathbf{s}(D) \end{aligned}$$

Es decir, se puede estimar la media de la población a partir de la media muestral. Un ejemplo de estadística no asociativa, es la *mediana*, en general no hay forma de combinar medianas muestrales para obtener la media de la población.

1.1.2.2. Observaciones univariadas

Usualmente, un objetivo preliminar del análisis de datos es describir el centroide y la dispersión de la distribución de una variable. Esto puede realizarse estimando la media μ y varianza σ^2 *poblacional* de una muestra de datos univariados $D = \{x_1, x_2, \dots, x_n\}$. Además del estimador de la media $\hat{\mu} = \sum_{i=1}^n x_i/n$, se requiere un estimador para la varianza $\hat{\sigma}^2$; por ejemplo, la diferencia entre las observaciones y la media muestral:

$$\begin{aligned}\hat{\sigma}^2 &= n^{-1} \sum_{i=1}^m (x_i - \hat{\mu})^2 \\ &= n^{-1} \sum_{i=1}^m x_i^2 - \left(n^{-1} \sum_{i=1}^m x_i \right)^2\end{aligned}$$

De lo anterior se deduce la estadística asociativa:

$$s(D)_{3 \times 1} = \begin{bmatrix} \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i^2 \\ n \end{bmatrix}$$

Para estimar la media y la varianza; sea $s(D) = [s_1, s_2, s_3]^T$, entonces los estimadores son:

$$\begin{aligned}\hat{\mu} &= \frac{s_1}{s_3} \\ \hat{\sigma}^2 &= \frac{s_2}{s_3} - \left(\frac{s_1}{s_3} \right)^2\end{aligned}$$

Esta estadística es asociativa porque la suma es asociativa; por ejemplo, $\sum_{i=1}^n x_i = \sum_{i=1}^m x_i + \sum_{i=m+1}^n x_i$ para $1 \leq m < n$. Por tanto, se tiene un algoritmo escalable para calcular los estimadores de media y varianza basado en la estadística asociativa $s(D)$ y en los modelos previos. Si la cantidad de datos D es muy grande para un sólo nodo, entonces se puede particionar como D_1, \dots, D_r y distribuirlos en r nodos para después utilizar el desarrollo anterior.

Tarea

Observaciones multivariadas

Sea un conjunto de datos consistente de n observaciones $D = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$, donde cada una es un vector de p variables $\mathbf{x}_i = [x_{i,1}, \dots, x_{i,p}]^T$; suponiendo que son resultado de vectores aleatorios multivariados $\mathbf{X}_i = [X_1, \dots, X_p]^T$ con esperanza $E(\mathbf{X}) = \boldsymbol{\mu}$ y matriz de varianzas $\text{var}(\mathbf{X}) = \boldsymbol{\Sigma}$, donde:

$$\boldsymbol{\mu}_{p \times 1} = [E(X_1), \dots, E(X_p)]^T = [\mu_1, \dots, \mu_p]^T$$

y

$$\boldsymbol{\Sigma}_{p \times p} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1p} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \cdots & \sigma_p^2 \end{bmatrix}$$

Los elementos de la diagonal $\sigma_1^2, \dots, \sigma_p^2$ son las varianzas de cada variable individual y los elementos fuera de la diagonal se conocen como covarianzas que describen la intensidad de la relación entre cada par de variables. El coeficiente de correlación poblacional:

$$\rho_{jk} = \frac{\sigma_{jk}}{\sigma_j \sigma_k} = \rho_{kj}$$

cuantifica la intensidad **lineal** de la asociación entre la j -ésima y k -ésima variable; está dentro del intervalo $[-1, 1]$. Valores cercanos a 1 ó -1 indican una relación casi lineal entre ellas; valores positivos indican una asociación positiva entre ambas, mientras que valores negativos son indicio de una relación negativa. Valores cercanos a 0 indican que de existir una relación entre las variables es no lineal. La matriz de correlaciones se determina como:

$$\rho_{p \times p} = \begin{bmatrix} 1 & \rho_{12} & \cdots & \rho_{1p} \\ \rho_{21} & 1 & \cdots & \rho_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{p1} & \rho_{p2} & \cdots & 1 \end{bmatrix}$$

Muestra que los siguientes estimadores definen algoritmos escalables:

◇ Estimador de μ a partir de la media muestral:

$$\bar{\mathbf{x}}_{p \times 1} = n^{-1} [\sum_{i=1}^n x_{i,1}, \dots, \sum_{i=1}^n x_{i,p}]^T$$

◇ Estimador de varianza muestral:

$$\hat{\sigma}_j^2 = n^{-1} \sum_{i=1}^n x_{i,j}^2 - \bar{x}_j^2$$

◇ Estimador de covarianza entre las variables j y k :

$$\hat{\sigma}_{jk} = n^{-1} \sum_{i=1}^n x_{i,j} x_{i,k} - \bar{x}_j \bar{x}_k$$

Nota: para la varianza y covarianza muestrales se elige n en lugar de $n - 1$ para obtener una estadística asociativa y para simplificar los cálculos, además para valores *suficientemente grandes*, la diferencia es despreciable.

◇ Estimador de Σ :

$$\hat{\Sigma} = n^{-1} \mathbf{M} - \bar{\mathbf{x}} \bar{\mathbf{x}}^T$$

donde \mathbf{M} es la *matriz de momentos*:

$$\mathbf{M}_{p \times p} = \begin{bmatrix} \sum x_{i,1}^2 & \sum x_{i,1} x_{i,2} & \cdots & \sum x_{i,1} x_{i,p} \\ \sum x_{i,2} x_{i,1} & \sum x_{i,2}^2 & \cdots & \sum x_{i,2} x_{i,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sum x_{i,p} x_{i,1} & \sum x_{i,p} x_{i,2} & \cdots & \sum x_{i,p}^2 \end{bmatrix}$$

y $\bar{\mathbf{x}} \bar{\mathbf{x}}^T$ es el producto externo del vector $\bar{\mathbf{x}}$ con su transpuesta:

$$\bar{\mathbf{x}} \bar{\mathbf{x}}^T_{p \times p} = \bar{\mathbf{x}}_{p \times 1} \bar{\mathbf{x}}^T_{1 \times p} = \begin{bmatrix} \bar{x}_1^2 & \cdots & \bar{x}_1 \bar{x}_p \\ \vdots & \ddots & \vdots \\ \bar{x}_p \bar{x}_1 & \cdots & \bar{x}_p^2 \end{bmatrix}$$

◇ Estimador de ρ :

$$\mathbf{R}_{p \times p} = \begin{bmatrix} 1 & r_{12} & \cdots & r_{1p} \\ r_{21} & 1 & \cdots & r_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ r_{p1} & r_{p2} & \cdots & 1 \end{bmatrix}$$

donde:

$$r_{jk} = \frac{\sum (x_{i,j} - \bar{x}_j)(x_{i,k} - \bar{x}_k)}{\hat{\sigma}_j \hat{\sigma}_k}$$

◇ Demuestra que $\mathbf{R} = \mathbf{D}\hat{\Sigma}\mathbf{D}$, con:

$$\mathbf{D} = \begin{bmatrix} \hat{\sigma}_1^{-1} & 0 & \cdots & 0 \\ 0 & \hat{\sigma}_2^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \hat{\sigma}_p^{-1} \end{bmatrix}$$

*

1.1.2.3. La matriz de momentos aumentada

El cálculo de $\bar{\mathbf{x}}$, $\hat{\Sigma}$ y \mathbf{R} puede acelerarse si se aumentan los vectores de datos insertando un 1 al inicio de cada uno; obteniendo a partir de estos vectores la *matriz de momentos aumentada*.

El i -ésimo vector aumentado es:

$$\mathbf{w}_i_{(p+1) \times 1} = \begin{bmatrix} 1 \\ \mathbf{x}_i \\ \mathbf{p} \times 1 \end{bmatrix} = \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,p} \end{bmatrix}$$

El producto externo de \mathbf{w}_i consigo mismo es una matriz de orden $(p+1) \times (p+1)$:

$$\mathbf{w}_i \mathbf{w}_i^T = \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,p} \end{bmatrix} \begin{bmatrix} 1 & x_{i,1} & \cdots & x_{i,p} \end{bmatrix} = \begin{bmatrix} 1 & x_{i,1} & \cdots & x_{i,p} \\ x_{i,1} & x_{i,1}^2 & \cdots & x_{i,1}x_{i,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i,p} & x_{i,p}x_{i,1} & \cdots & x_{i,p}^2 \end{bmatrix}$$

Sumando n productos externos, obtenemos la matriz aumentada \mathbf{A} :

$$\mathbf{A}_{(p+1) \times (p+1)} = \sum \mathbf{w}_i \mathbf{w}_i^T = \begin{bmatrix} n & \sum x_{i,1} & \sum x_{i,2} & \cdots & \sum x_{i,p} \\ \sum x_{i,1} & \sum x_{i,1}^2 & \sum x_{i,1}x_{i,2} & \cdots & \sum x_{i,1}x_{i,p} \\ \sum x_{i,2} & \sum x_{i,2}x_{i,1} & \sum x_{i,2}^2 & \cdots & \sum x_{i,2}x_{i,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{i,p} & \sum x_{i,p}x_{i,1} & \sum x_{i,p}x_{i,2} & \cdots & \sum x_{i,p}^2 \end{bmatrix}$$

La matriz de momentos aumentada \mathbf{A} difiere de la matriz de momentos \mathbf{M} al tener una fila y columna adicional en la parte superior e izquierda que contiene n y la suma de cada variable; por tanto \mathbf{A} puede expresarse como:

$$\mathbf{A} = \begin{bmatrix} n & n\bar{\mathbf{x}}^T \\ 1 \times 1 & 1 \times p \\ n\bar{\mathbf{x}} & \mathbf{M} \\ p \times 1 & p \times p \end{bmatrix}$$

La matriz de momentos aumentada es una estadística asociativa y apartir de \mathbf{A} , se extraen directamente \mathbf{M} y $\bar{\mathbf{x}}$; con éstos, se puede obtener la matriz de varianzas $\hat{\mathbf{\Sigma}} = n^{-1}\mathbf{M} - \bar{\mathbf{x}}\bar{\mathbf{x}}^T$. Así mismo, \mathbf{D} y \mathbf{R} puede ser fácilmente calculada.

En un entorno distribuido, se crea una partición D_1, D_2, \dots, D_r del conjunto de datos D y cada subconjunto se procesa independientemente. El subconjunto D_j entrega una matriz \mathbf{A}_j y las matrices $\mathbf{A}_1 \dots \mathbf{A}_r$ se combinan al final de los r procesos para obtener \mathbf{A} de la siguiente manera:

$$\mathbf{A} = \sum_{j=1}^r \mathbf{A}_j$$

Finalmente \mathbf{M} y $\bar{\mathbf{x}}$ se extraen de \mathbf{A} y se calculan $\hat{\mathbf{\Sigma}}$, \mathbf{D} y \mathbf{R} .

Tarea

Escribir un código que obtenga la matriz de momentos aumentada a partir de un conjunto de vectores de datos.

$$\mathbf{A}_{(p+1) \times (p+1)} = \sum \mathbf{w}_i \mathbf{w}_i^T = \begin{bmatrix} n & \sum x_{i,1} & \sum x_{i,2} & \cdots & \sum x_{i,p} \\ \sum x_{i,1} & \sum x_{i,1}^2 & \sum x_{i,1}x_{i,2} & \cdots & \sum x_{i,1}x_{i,p} \\ \sum x_{i,2} & \sum x_{i,2}x_{i,1} & \sum x_{i,2}^2 & \cdots & \sum x_{i,2}x_{i,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{i,p} & \sum x_{i,p}x_{i,1} & \sum x_{i,p}x_{i,2} & \cdots & \sum x_{i,p}^2 \end{bmatrix}$$

*

1.1.2.4. Histogramas

Un histograma sirve para representar e interpretar estadísticas visualmente; además, el histograma se puede construir como una estadística asociativa y, por lo tanto, es útil en el análisis de conjuntos muy grandes de datos. Cuando el volumen de datos crece demasiado, los histogramas son la opción para visualizarlos porque pueden mostrar la distribución con muy buen detalle.

Construcción de histogramas

Un histograma es un conjunto de pares, cada par corresponde con cada uno de los rectángulos mostrados. Un elemento del par es el intervalo que define la base del rectángulo y el segundo elemento especifica la altura. Dicha altura puede ser el número de observaciones que caen dentro del intervalo o la frecuencia relativa de las observaciones en ese mismo intervalo. La unión de los intervalos abarca el rango de la variable de interés. Un histograma se define matemáticamente como un conjunto de pares

$$H = \{(b_1, p_1), \dots, (b_h, p_h)\}$$

El número de intervalos es h ; el primero es $b_i = [l_1, u_1]$ y para $i > 1$, el i -ésimo intervalo es $b_i = (l_i, u_i]$. El segundo elemento de cada pareja, p_i es la frecuencia relativa de las observaciones que pertenecen a ese intervalo; este término se puede utilizar como un estimador de la proporción de la población que pertenece a b_i . El siguiente intervalo (b_{i+1}) toma como límite inferior (l_{i+1}) el límite superior anterior (u_i), i.e. $l_{i+1} = u_i$. En general, todas las observaciones del conjunto de datos pertenecen a la base $[l_1, u_h]$ del histograma.

Supongamos que el conjunto de datos es tan grande que no puede almacenarse en memoria; entonces, se requiere un algoritmo escalable para contruir el histograma H . En resulmen, H se contruye contando el número de observaciones de cada intervalo; esto implica que los datos se procesarán dos veces: en la primera pasada se determina el número y el tamaño de cada intervalo, en la segunda se obtiene el número de observaciones de cada intervalo.

Un algoritmo que ayude a contruir un histograma comienza con la obtención de la mínima ($x(1)$) y máxima ($x(n)$) observacion en D . La primera pasada sobre los datos permite obtener el rango de valores ($x(n) - x(1)$); y la anchura de cada intervalo $w = x(n) - x(1)/h$, donde h es el número de intervalos deseado. Los intervalos obtenidos son:

$$\begin{aligned} b_1 &= [x_1, x_1 + w] \\ &\vdots \\ b_i &= (x_1 + (i-1)w, x_1 + (i)w] \\ &\vdots \\ b_h &= (x_1 + (h-1)w, x_n] \end{aligned}$$

Es decir, se crea un mapeo del conjunto de datos D a un conjunto de intervalos (contenedores, *bins*) $B = \{b_1, \dots, b_h\}$; escrito como $D \mapsto B$.

La segunda parte del algoritmo mapea D y B a un diccionario C en el cual las llaves son los intervalos y los valores son el total de observaciones contenidas en cada intervalo; se escribe $(D, B) \mapsto C$.

Desde el punto de vista computacional, esta segunda parte crea el diccionario C contando el número de observaciones de cada intervalo; cuando se encuentra el intervalo al que pertenece una observación, se incrementa el contador de dicho intervalo. Una vez procesadas todas las observaciones, se puede obtener la frecuencia relativa de observaciones de cada intervalo diviendo su contador por n .

Si el algoritmo será escalable, entonces la estadística con la que se construye H debe ser asociativa; la clave para la asociatividad y escalabilidad es que se utiliza sólo un conjunto de intervalos como base de H .

Como se describió, con la primera pasada se obtiene el vector de dos elementos $\mathbf{s}(D) = [\min(D), \max(D)]^T$. Suponiendo que D_1, D_2, \dots, D_r es una partición de D , con

$$\mathbf{s}(D_j) = \begin{bmatrix} \min(D_j) \\ \max(D_j) \end{bmatrix}, j = 1, \dots, r$$

Sea $s_1(D) = \min(D)$ y $s_2(D) = \max(D)$, tales que $\mathbf{s}(D) = [s_1(D), s_2(D)]^T$; entonces,

$$\begin{aligned} s_1(D) &= \min(D) \\ &= \min\{D_1 \cup D_2 \cup \dots \cup D_r\} \\ &= \min\{\min(D_1), \dots, \min(D_r)\} \\ &= \min\{s_1(D_1), \dots, s_1(D_r)\} \end{aligned}$$

De forma similar, $\max(D) = \max\{\max(D_1), \dots, \max(D_r)\}$, por tanto $s_2(D) = \max\{s_2(D_1), \dots, s_2(D_r)\}$. Como $\mathbf{s}(D)$ puede obtenerse a partir de $\mathbf{s}(D_1), \dots, \mathbf{s}(D_h)$, la estadística \mathbf{s} es asociativa.

La segunda parte del algoritmo se encarga de llenar el diccionario $C = \{(b_1, c_1), \dots, (b_h, c_h)\}$ determinando el valor c_j como la cuenta de observaciones del conjunto de datos D que caen dentro del intervalo b_j . Matemáticamente, $(D_j, B) \mapsto C_j$; como los conjuntos C_1, \dots, C_r se construyen agregando a los contadores de cada intervalo y como la suma es asociativa; entonces es factible tener un algoritmo escalable para la construcción de histogramas.

Tarea

- ◇ Obtener las expresiones para determinar los intervalos de contenedores para histogramas bidimensionales en función del número de contenedores (*bins*); así como la forma de determinar el total de observaciones de cada contenedor

*

Ejemplos

Existen varias formas de realizar histogramas en Python:

Pandas

Importando bibliotecas:

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

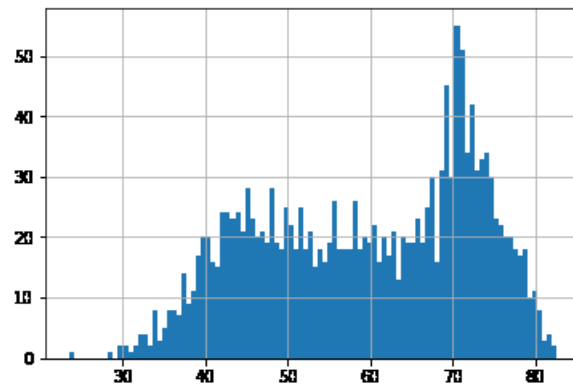
Datos:

```
data_url='https://bit.ly/2cLzoxH'
brecha=pd.read_csv(data_url)
brecha.head(n=3)
```

	country	year	pop	continent	lifeExp	gdpPercap
0	Afghanistan	1952	8425333.0	Asia	28.801	779.445314
1	Afghanistan	1957	9240934.0	Asia	30.332	820.853030
2	Afghanistan	1962	10267083.0	Asia	31.997	853.100710

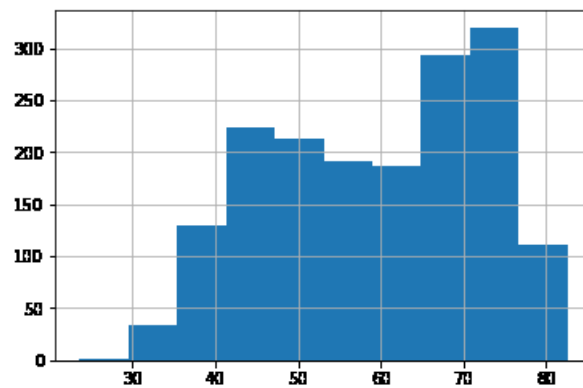
Pandas incluye el método *hist*; haciendo el histograma de la esperanza de vida:

```
brecha['lifeExp'].hist(bins=100)
```



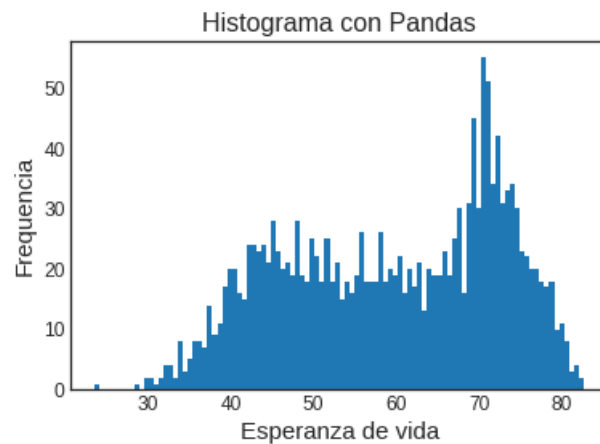
Es recomendable probar con diferentes valores para el número de contenedores (*bins*):

```
brecha['lifeExp'].hist(bins=10)
```



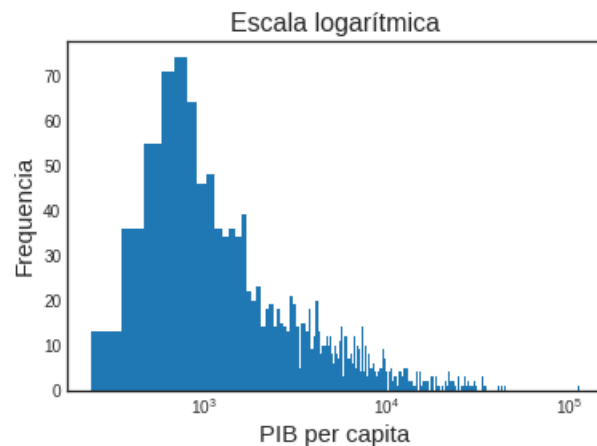
Se observa que la reducción tan abrupta hace que se pierdan algunos detalles de la distribución, con valores grandes se pueden ver más patrones que se ocultan con valores pequeños. Se puede agregar información y dar formato al histograma:

```
brecha['lifeExp'].hist(bins=100, grid=False, xlabelsize=12, ylabelsize=12)
plt.title("Histograma con Pandas", fontsize=16)
plt.xlabel("Esperanza de vida", fontsize=15)
plt.ylabel("Frecuencia", fontsize=15)
```



También es posible utilizar escalas logarítmicas; aplicado al eje x en el histograma del ingreso producto interno bruto *per cápita*:

```
brecha['gdpPercap'].hist(bins=1000,grid=False)
plt.title('Escala logarítmica', fontsize=16)
plt.xlabel("PIB per capita", fontsize=15)
plt.ylabel("Frecuencia", fontsize=15)
plt.xscale('log')
```



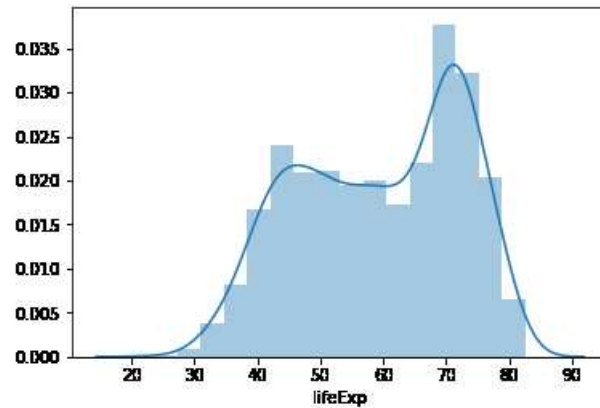
Seaborn

Esta biblioteca también tiene incluida una función para histogramas.

```
import seaborn as sns
```

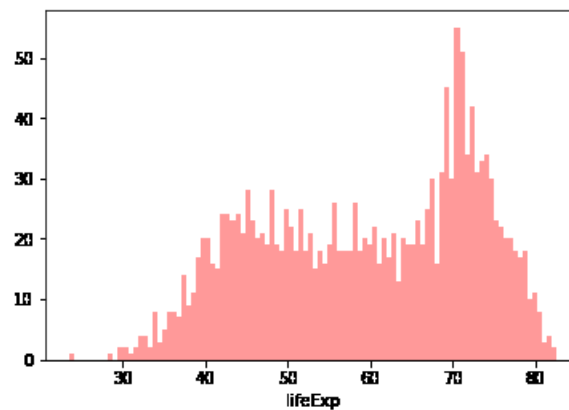
distplot que toma una columna de un *DataFrame* de Pandas como argumento y por omisión incluye la curva de *Kernel Density Estimation* (kde):

```
sns.distplot(brecha['lifeExp'])
```



Por default, el número de contenedores se calcula de acuerdo al total de valores; puede establecerse con el parámetro *bins*, y también puede eliminarse la curva kde:

```
sns.distplot(brecha['lifeExp'], kde=False, color='red', bins=100)
```



Y también puede añadirse información y formato:

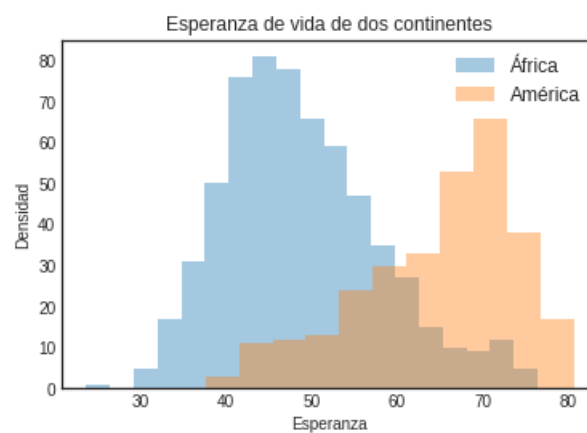
```
sns.distplot(brecha['lifeExp'], kde=False, color='red', bins=100)
plt.title('Histograma con Seaborn', fontsize=16)
plt.xlabel('Esperanza de vida', fontsize=15)
plt.ylabel('Frecuencia', fontsize=15)
```



También se pueden mostrar múltiples histogramas:

```
df = brecha[brecha.continent == 'Africa']
sns.distplot(df['lifeExp'], kde=False, label='Africa')
df = brecha[brecha.continent == 'Americas']
sns.distplot(df['lifeExp'], kde=False, label='América')

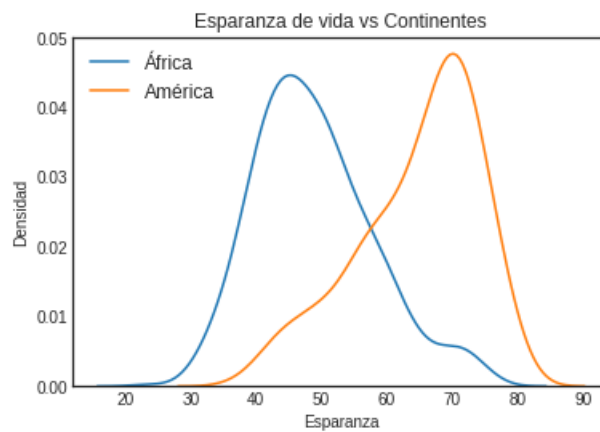
plt.legend(prop={'size': 12})
plt.title('Esperanza de vida de dos continentes')
plt.xlabel('Esperanza')
plt.ylabel('Densidad')
```



Solamente las curvas kde de ambos:

```
df = brecha[brecha.continent == 'Africa']
sns.distplot(df['lifeExp'], hist = False, kde = True, label='Africa')
df = brecha[brecha.continent == 'Americas']
sns.distplot(df['lifeExp'], hist = False, kde = True, label='Americas')

plt.legend(prop={'size': 12})
plt.title('Esparanza de vida vs Continentes')
plt.xlabel('Esparanza') plt.ylabel('Densidad')
```

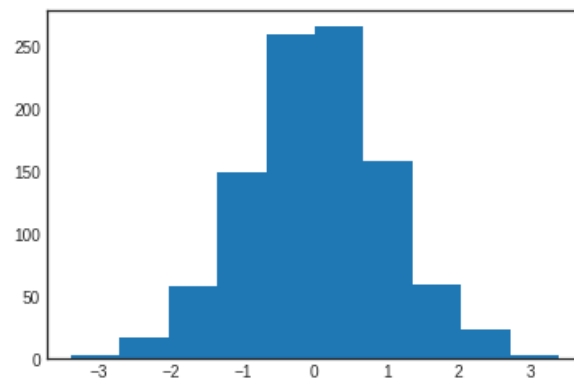


Matplotlib

También incluye capacidades para realizar histogramas.

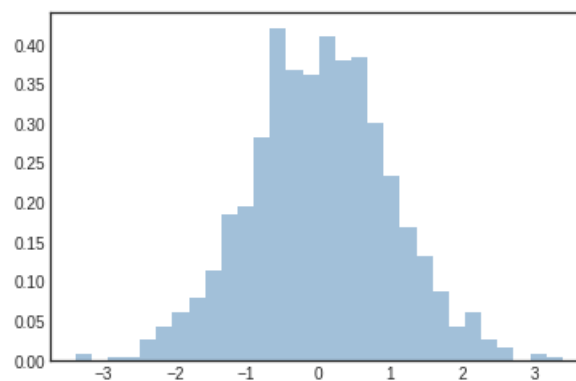
```
import numpy as np
import matplotlib.pyplot as plt
```

```
plt.style.use('seaborn-white')
data = np.random.randn(1000)
plt.hist(data)
```



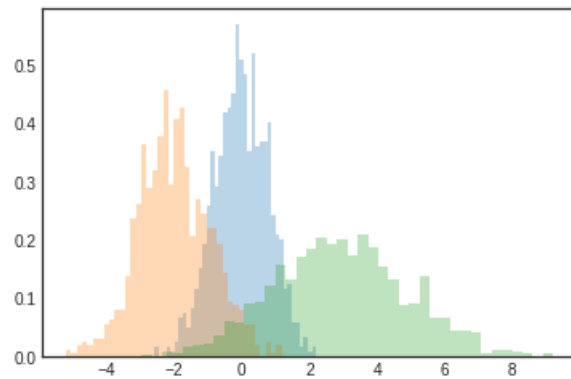
Formateo:

```
plt.hist(data, bins=30, density=True, alpha=0.5,  
         histtype='stepfilled', color='steelblue',  
         edgecolor='none')
```



Al igual que con seaborn, es posible desplegar varios conjuntos de datos:

```
x1 = np.random.normal(0, 0.8, 1000)  
x2 = np.random.normal(-2, 1, 1000)  
x3 = np.random.normal(3, 2, 1000)  
# formato  
dic = dict(histtype='stepfilled', alpha=0.3, bins=40, density=True)  
  
plt.hist(x1, **dic)  
plt.hist(x2, **dic)  
plt.hist(x3, **dic)
```



Es posible calcular el histograma sin desplegarlo; es decir, obtener el total de elementos en cada contenedor, se puede hacer:

```
total, bordes = np.histogram(data, bins=5)
print(total)
print(bordes)
```

Histogramas bidimensionales

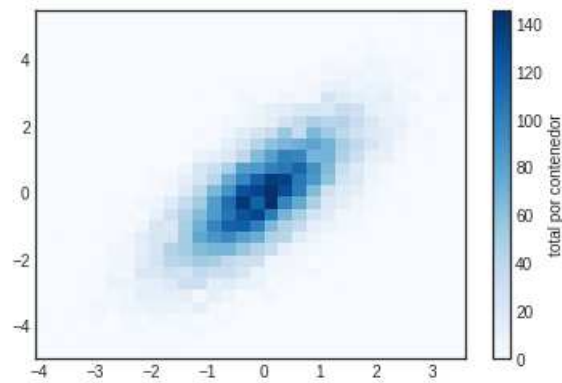
Es común tener datos correlacionados y se puede obtener un histograma que los muestre.

Matplotlib

Incluye el método *hist2d*:

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T

plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('total por contenedor')
```

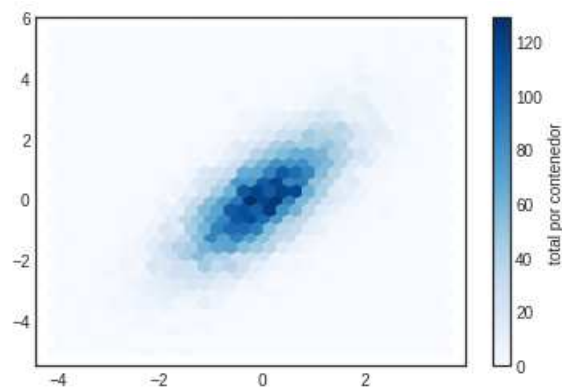


También se puede obtener el total de elementos en cada contenedor:

```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

Se pueden utilizar contenedores hexagonales:

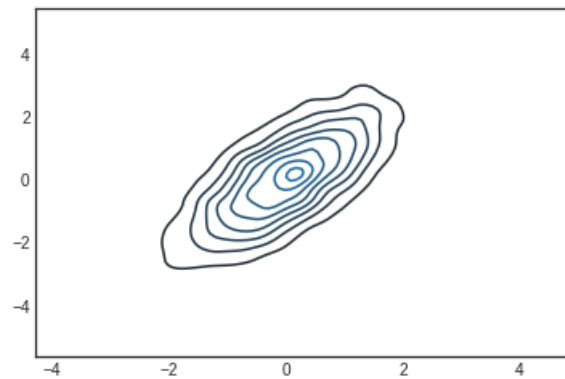
```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='total por contenedor')
```



Seabron

Incluye una función para obtener la curva *kde* bidimensional:

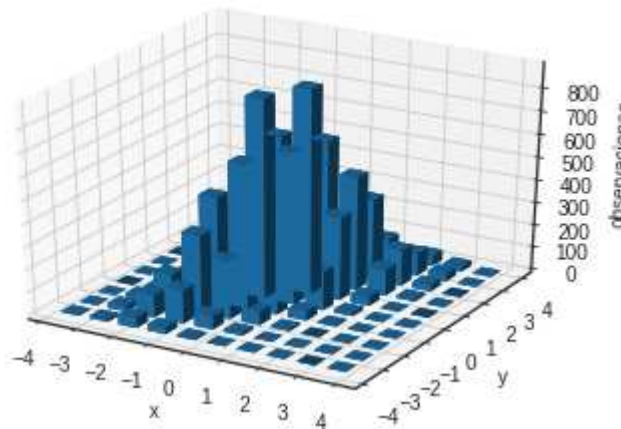
```
data = np.random.multivariate_normal(mean, cov, 10000)
sns.kdeplot(data)
```



Histogramas bidimensionales: vista en 3D

Para obtener una vista en 3D, se deben almacenar los valores que devuelve *hist2D* y crear una gráfica con ayuda de *bar3d*:

```
import matplotlib.pyplot as plt
import numpy as np
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x,y = np.random.multivariate_normal(mean, cov, 10000).T
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# Histograma y contenedores bidimensionales
hist, xedges, yedges = np.histogram2d(x, y, bins=10, range=[[-4, 4], [-4, 4]])
# Arreglos para las posiciones de las barras
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing='ij')
xpos = xpos.ravel()
ypos = ypos.ravel()
zpos = 0
# Arreglos con la dimensión de las barras
dx = dy = 0.5 * np.ones_like(zpos)
dz = hist.ravel()
ax.bar3d(xpos, ypos, zpos, dx, dy, dz, zsort='average')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('observaciones')
plt.show()
```



1.1.3. Introducción a *Hadoop*, *MapReduce* y procesos por lotes

Como ya se ha visto, cuando el volumen de datos excede las capacidades de una sola computadora, una solución es distribuir los datos en una red en la que cada nodo procesa subconjuntos de los datos para finalmente combinarlos para completar la tarea. Para que esta solución sea exitosa, el algoritmo debe cumplir cierta estructura y la red debe ser administrada: el entorno *Hadoop* y el diseño con *MapReduce* proveen estas características. Hadoop es una colección de programas y servicios que construyen el cluster, distribuyen los datos y controlan su procesamiento y transmisión de los resultados. El diseño de programas con MapReduce asegura la escalabilidad y ésta a su vez asegura que los resultados sean independientes de la configuración del cluster.

La *Apache Software Foundation* es una organización que mantiene y organiza los desarrollos de proyectos *open-source* relacionados con Hadoop. La arquitectura de Hadoop contiene gran variedad de métodos para cómputo distribuido. Puede decirse que Hadoop es una colección de proyectos *open-source* de Apache administrados y controlados por su fundación.

Un *cluster* es un conjunto de dos o más computadoras conectadas por una red de alta velocidad. En Hadoop los nodos pueden clasificarse como *maestro* (*master*) o *trabajador* (*worker*). El *NameNode* es el maestro y tiene a su cargo el control mediante dos subsistemas: El ***Hadoop File Distributed System*** (HDFS) y el **sistema de asignación y administración**, conocido como YARN (*Yet Another Resource Negotiator*). En YARN, el nodo maestro asigna procesos a los nodos trabajadores con el objetivo de optimizar el uso de los recursos del cluster. Los procesos que se llevan a cabo en un *DataNode* tienen su propio *administrador de nodo*.

1.1.3.1. *Hadoop Distributed File System* (HDFS)

El propósito del HDFS es particionar y distribuir los datos por todo el cluster como una colección de bloques (subconjuntos). El cliente o usuario, se comunica con HDFS a través de una *application program interface* (API). La distribución de los bloques de datos y el monitoreo de los *DataNodes* se lleva a cabo en una computadora maestra, llamada *NameNode*. HDFS almacena un directorio en el nodo maestro que contiene todos los archivos y las ubicaciones (*DataNodes*) en las que se almacenan los datos y no almacena ningún dato a procesar en este *NameNode*. La probabilidad de un error fatal en algún *DataNode* es muy alta si el número de nodos es grande; por tanto, se tienen redundancias en el cluster al distribuir cada bloque de datos en varios *DataNodes*.

Si un nodo falla, otros nodos realizan el procesamiento asignado al nodo fallido. El nodo primario *NameNode* tiene un respaldo por un *NameNode* secundario por si falla el primero.

En la figura 1.4 se observa una versión pequeña del trabajo de HDFS. Se encuentra el *Node-Name* principal y el secundario; hay cuatro *DataNodes* y dos bloques son repartidos en ellos, cada bloque es enviado a dos nodos para tener respaldo.

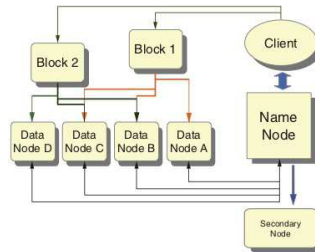


Figura 1.4: Ejemplo sencillo de la arquitectura de Hadoop

1.1.3.2. *MapReduce*

El procesamiento de datos con Hadoop debe realizarse de acuerdo con una estructura particular conocida como MapReduce, que consiste de tres etapas secuenciales. En la primera (*mapping*) se mapean los datos a un conjunto de pares clave-valor; después, la etapa de *shuffle* acomoda los pares clave-valor de forma que todas las instancias de una clave particular se localicen en un mismo *DataNode*; la tercera es la etapa de reducción (*reducer*) en la que los datos se reducen a un conjunto de estadísticas, o listas, tablas o alguna otra forma de resumen concebida por el analista. El analista/científico de datos crea el *mapper* y el *reducer* por ejemplo como *scripts* de Python; la etapa de *shuffle* se realiza internamente por Hadoop.

Mapping

Es el primer paso en el algoritmo de MapReduce, su propósito principal es organizar los datos para la reducción. En cada *DataNode*, los registros se mapean a pares clave-valor; como los diccionarios estudiados anteriormente. Las claves permiten que el algoritmo MapReduce sea escalable; por tanto, son esenciales para el procesamiento óptimo de grandes conjuntos de datos. La elección de las llaves es responsabilidad del científico de datos y deben ser consistentes con los objetivos del análisis requerido. Adelantando un poco, todos los registros asociados a una clave particular, se reduzcan juntos y ningún valor asociado a diferentes llaves se reducirán juntos.

La importancia de las llaves se observa en la siguiente etapa de MapReduce: *shuffle*. Este proceso mueve las parejas clave-valor en el cluster de forma que todas las parejas con una llave particular se localicen en un mismo *DataNode*. Por ejemplo, supongamos que k_i es la i -ésima llave y que existen n_i observaciones asociadas a esa llave; entonces todas esas observaciones se escribirán en un mismo nodo. En la etapa de reducción, el *reducer* concentrará todas las n_i observaciones asociadas a k_i de acuerdo a las instrucciones escritas en su algoritmo.

En general, el número de registros generados por el mapeo distribuidos en todos los bloques es el mismo que el total inicial de registros; sin embargo, puede realizar operaciones diferentes a generar las parejas llave-valor; por ejemplo, eliminar registros o ignorar variables que no se utilizan en el algoritmo de reducción.

Reduction

El papel que juega el algoritmo de reducción es simplificar los datos a una forma útil de interpretación. Comúnmente, el *reducer* tiene como salida un resumen estadístico para cada llave.

Es importante recalcar que cuando no existe una división natural en los datos, la estadística computada por el reductor debe ser asociativa; de otra forma no hay garantía de que la salida pueda agregarse de forma óptima y bien determinada. En este último caso, el algoritmo no será escalable aún cuando se utilicen Hadoop y MapReduce.

Ejemplo: hola mundo

Este ejemplo muestra los códigos para hacer un contador de palabras que puede ser ejecutado con la arquitectura de *Hadoop*.

Primero, se escribe el código del *mapper*:

```
#!/usr/bin/env python
"""mapper.py"""
import sys

# los datos se leen de la entrada estándar
for line in sys.stdin:
    # Eliminar espacios en blanco
    line = line.strip()
    # separar "palabras"
    words = line.split()
    # incrementar contadores
    for word in words:
        # escribir el resultado a la salida estándar
        # esta salida será la entrada para el reducer.py
        # separado por tabulador, cada palabra cuenta 1
        print('%s\t%s' % (word, 1))
```

Por ejemplo, en la terminal:

```
$ echo "hola mundo cruel probando hola mundo adiós mundo cruel" \  
| ./mapper.py  
hola      1  
mundo     1  
cruel     1  
probando          1  
hola      1  
mundo     1  
adiós     1  
mundo     1  
cruel     1
```

Ahora el *reducer*:

```
#!/usr/bin/env python  
"""reducer.py"""  
import sys  
  
current_word = None  
current_count = 0  
word = None  
for line in sys.stdin:  
    line = line.strip()  
    # separar los datos obtenidos de mapper.py  
    word, count = line.split('\t', 1)  
    # convertir la cuenta a entero  
    try:  
        count = int(count)  
    except ValueError:  
        # si no era un entero, se ignora  
        continue  
    # este IF funciona porque Hadoop ordena la salida del mapper  
    # por la clave (word) antes de que la reciba el reducer  
    if current_word == word:  
        current_count += count  
    else:  
        if current_word:  
            print ('%s\t%s' % (current_word, current_count))  
            current_count = count  
            current_word = word  
# es necesario escribir la última palabra  
if current_word == word:  
    print ('%s\t%s' % (current_word, current_count))
```

En la terminal (nótese el uso de *sort* para simular el trabajo interno de Hadoop):

```
$ echo "hola mundo cruel probando hola mundo adiós mundo cruel" \  
| ./mapper_0.py | sort -k1,1 | ./reducer_0.py  
adiós      1  
cruel      2  
hola       2  
mundo      3  
probando           1
```

Este ejemplo se puede mejorar usando iteradores y generadores de Python; el *mapper*:

```
#!/usr/bin/env python  
"""Un Mapper más avanzado con iteradores y generadores de Python"""  
import sys  
def read_input(file):  
    for line in file:  
        yield line.split()  
def main(separator='\t'):  
    data = read_input(sys.stdin)  
    for words in data:  
        for word in words:  
            print ('%s%s%d' % (word, separator, 1))  
if __name__ == "__main__":  
    main()
```

El *reducer*:

```
#!/usr/bin/env python
"""Un Reducer más avanzado con iteradores y generadores de Python"""
from itertools import groupby
from operator import itemgetter
import sys

def read_mapper_output(file, separator='\t'):
    for line in file:
        yield line.rstrip().split(separator, 1)

def main(separator='\t'):
    data = read_mapper_output(sys.stdin, separator=separator)
    # groupby agrupa múltiples contadores por palabra y crea un iterador
    # que devuelve claves consecutivas y su grupo asociado
    # current_word - la clave (palabra)
    # group - iterador que entrega todas las parejas [<current_word>, <count>]
    for current_word, group in groupby(data, itemgetter(0)):
        try:
            total_count = sum(int(count) for current_word, count in group)
            print ("%s%s%d" % (current_word, separator, total_count))
        except ValueError:
            pass
if __name__ == "__main__":
    main()
```

El *Project Gutenberg* cuenta con gran cantidad de libros con el texto plano disponible para probar nuestro ejemplo: <https://www.gutenberg.org/>

Alice's Adventures in Wonderland: <https://www.gutenberg.org/files/11/11-0.txt>

Para probarlo en una arquitectura real de Hadoop, se puede hacer con el servicio de *Google Cloud Platform*, siguiendo los pasos descritos en:

<https://github.com/GoogleCloudPlatform/solutions-google-compute-engine-cluster-for-hadoop>

Además hay un video que explica la forma de ejecutarlo:

<https://www.youtube.com/watch?v=se9vV8eIZME>

Tarea

Escribir un código que simule la obtención la matriz de momentos aumentada a partir de un conjunto de vectores de datos utilizando MapReduce.

$$\mathbf{A}_{(p+1) \times (p+1)} = \sum \mathbf{w}_i \mathbf{w}_i^T = \begin{bmatrix} n & \sum x_{i,1} & \sum x_{i,2} & \cdots & \sum x_{i,p} \\ \sum x_{i,1} & \sum x_{i,1}^2 & \sum x_{i,1}x_{i,2} & \cdots & \sum x_{i,1}x_{i,p} \\ \sum x_{i,2} & \sum x_{i,2}x_{i,1} & \sum x_{i,2}^2 & \cdots & \sum x_{i,2}x_{i,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{i,p} & \sum x_{i,p}x_{i,1} & \sum x_{i,p}x_{i,2} & \cdots & \sum x_{i,p}^2 \end{bmatrix}$$

*