



PYTHON PARA LA INTELIGENCIA ARTIFICIAL

D. Iván Fuertes Torrecilla

MÁSTER UNIVERSITARIO EN INTELIGENCIA
ARTIFICIAL



Universidad
Internacional
de Valencia

Este material es de uso exclusivo para los alumnos de la Universidad Internacional de Valencia. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la Universidad Internacional de Valencia, sin autorización expresa de la misma.

Edita

Universidad Internacional de Valencia

Máster Universitario en
Inteligencia Artificial

Python para la inteligencia artificial
6 ECTS

D. Iván Fuertes Torrecilla

Leyendas



Enlace de interés



Ejemplo



Importante



abc Los términos resaltados a lo largo del contenido en color **naranja** se recogen en el apartado **GLOSARIO**.

Índice

| | |
|---|----|
| CAPÍTULO 1. INTRODUCCIÓN | 9 |
| 1.1. Contexto..... | 9 |
| 1.2. Por qué Python..... | 10 |
| 1.3. Convenciones..... | 11 |
| 1.4. Librerías más populares y útiles | 12 |
| 1.5. Instalación | 12 |
| 1.6. Python 2 vs. Python 3 | 13 |
| 1.7. Contenidos del curso | 13 |
| | |
| CAPÍTULO 2. PYTHON 101..... | 14 |
| 2.1. Intérprete básico | 14 |
| 2.2. Ejecución de scripts | 15 |
| 2.3. Jupyter Notebook | 16 |
| 2.4. Sintaxis básica..... | 17 |
| 2.4.1. Variables..... | 17 |
| 2.4.2. Comentarios..... | 19 |
| 2.4.3. Comparaciones condicionales | 19 |
| 2.4.4. Loops..... | 20 |
| 2.4.5. Funciones..... | 22 |
| 2.5. Colecciones en Python | 26 |
| 2.5.1. Tuplas | 26 |
| 2.5.2. Listas | 26 |
| 2.5.3. Secuencias | 27 |
| 2.5.4. Diccionarios | 28 |
| 2.5.5. Sets (conjuntos) | 29 |
| 2.5.6. Python comprehension | 30 |
| | |
| CAPÍTULO 3. COLECCIONES: NUMPY | 32 |
| 3.1. Objeto básico en NumPy: ndarray..... | 32 |
| 3.1.1. Dimensiones..... | 33 |
| 3.1.2. Manipulación de ndarrays | 34 |
| 3.1.3. Índices y slicing | 35 |

| | |
|---|-----------|
| 3.2. Funciones matemáticas sobre colecciones | 36 |
| 3.3. Filtrado de datos..... | 37 |
| 3.4. Exploración y estadística descriptiva | 38 |
| 3.5. Álgebra lineal | 39 |
| 3.6. Valores aleatorios..... | 40 |
| | |
| CAPÍTULO 4. ESTRUCTURAS DE DATOS: PANDAS..... | 42 |
| 4.1. Series | 43 |
| 4.2. DataFrame..... | 46 |
| 4.3. Trabajo con Series y DataFrame..... | 48 |
| 4.3.1. Búsqueda..... | 48 |
| 4.3.2. Indexación y slicing | 49 |
| 4.3.3. Ordenación de Series y DataFrame..... | 51 |
| 4.4. Operaciones con Series y DataFrame | 52 |
| 4.5. Estadística con Series y DataFrame | 54 |
| | |
| CAPÍTULO 5. VISUALIZACIÓN: MATPLOTLIB | 56 |
| 5.1. Matplotlib 101 | 57 |
| 5.2. Múltiples gráficos | 59 |
| 5.3. Decoraciones y anotaciones..... | 61 |
| 5.4. Tipos de gráficos | 62 |
| 5.5. Grabación de gráficos a archivo | 63 |
| 5.6. Representación con Pandas | 63 |
| | |
| CAPÍTULO 6. PYTHON PARA CIENCIA DE DATOS..... | 66 |
| 6.1. Lectura y escritura de archivos | 66 |
| 6.1.1. En Python | 67 |
| 6.1.2. En NumPy | 69 |
| 6.1.3. En Pandas | 69 |
| 6.2. Trabajo con strings | 70 |
| 6.3. Limpieza de datos | 72 |
| 6.3.1. Aplicación de funciones a una colección..... | 72 |
| 6.3.2. Pandas para transformar datos..... | 73 |

| | |
|---|-----|
| CAPÍTULO 7. GENERADORES DE DATOS | 76 |
| 7.1. Generadores en Python..... | 76 |
| 7.1.1. Expresiones generadoras | 79 |
| 7.1.2. Crear pipelines de datos con generadores..... | 80 |
| 7.2. Generadores de Python para entrenar modelos | 84 |
| 7.3. Generadores de datos de imágenes con Keras..... | 85 |
| 7.3.1. Flow | 88 |
| 7.3.2. Augmentations | 88 |
| 7.4. Generadores de datos custom con Keras..... | 92 |
| | |
| CAPÍTULO 8. ANÁLISIS DE DATOS TEXTUALES CON NLTK..... | 97 |
| 8.1. Natural Language Toolkit NLTK | 98 |
| 8.2. Importar la librería NLTK y la herramienta de descargas | 98 |
| 8.3. Concordancia y colocaciones | 101 |
| 8.4. Analizar la frecuencia de las palabras..... | 103 |
| 8.5. Seleccionar palabras de un texto..... | 105 |
| 8.6. Bigramas y colocaciones | 106 |
| 8.7. Sinónimos y antónimos..... | 107 |
| 8.8. Stemming y lematización | 109 |
| 8.9. Texto en internet | 110 |
| 8.10. Extraer texto de páginas HTML | 112 |
| 8.11. Análisis de sentimiento | 113 |
| | |
| CAPÍTULO 9. RECONOCIMIENTO DE DÍGITOS ESCRITOS A MANO CON SCIKIT-LEARN..... | 117 |
| 9.1. El conjunto de datos de dígitos | 118 |
| 9.2. Aprendiendo y prediciendo | 120 |
| | |
| CAPÍTULO 10. ANÁLISIS DE IMÁGENES Y RECONOCIMIENTO DE CARAS CON OPENCV | 124 |
| 10.1. OpenCV..... | 125 |
| 10.2. Cargar y mostrar una imagen..... | 125 |
| 10.3. Trabajando con imágenes..... | 126 |
| 10.4. Operaciones elementales con imágenes | 127 |
| 10.5. Blending | 129 |
| 10.6. Detección de bordes..... | 129 |
| 10.6.1. Teoría del gradiente de imagen | 130 |
| 10.7. Detección de caras | 134 |

| | |
|---------------------------------|-----|
| GLOSARIO | 138 |
| ENLACES DE INTERÉS | 142 |
| BIBLIOGRAFÍA..... | 143 |



Capítulo 1

Introducción

1.1. Contexto

El aumento actual del interés en la inteligencia artificial (IA) se debe en gran medida a una mejora cualitativa en los algoritmos de *machine learning* (aprendizaje automático). Esta mejora ha sido posible por dos cambios fundamentales acontecidos en los últimos quince años: el incremento de la capacidad computacional de los ordenadores (gracias, en parte, al paralelismo de las GPU y a la computación en la nube) y la cantidad descomunal de datos disponibles para entrenar algoritmos (con todas las grandes empresas que ofrecen servicios a cambio de datos). Pero de nada sirve disponer de *petabytes* (millones de *gigabytes*) de información y tener a nuestro alcance computadoras en la exaescala (capaces de ejecutar billones de billones de operaciones por segundo) si no hablamos el lenguaje de las máquinas. En este sentido, cobra mucha relevancia dominar la programación por ordenador.

La IA está llamada a ser la cuarta revolución industrial (López, 2018), ya que es capaz de hacer cosas a una escala que antes era impensable. Y para formar parte de esta revolución altamente técnica, es necesario programar.

1.2. Por qué Python

Una pregunta frecuente que suele rondar la mente de quienes quieren iniciarse en el mundo de la IA es: ¿cuál es el mejor lenguaje de programación para la IA? Responder a esta pregunta normalmente requiere atender a varios niveles de complejidad, ya que puede referirse a qué lenguaje es cualitativamente mejor (algo casi imposible de evaluar) o a cuál es más sencillo o aplicable. Sin embargo, la mayor parte del tiempo, la pregunta tiene un tono más utilitario. Quienes se la formulan desean saber qué lenguaje tiene más adopción en la industria de la IA o, dicho de otro modo, con cuál se le van a abrir más puertas.

Dejando de lado las discusiones cuasifilosóficas, hay muchos lenguajes de programación adecuados para la IA. Desde los generalistas C, C++ y Java a los más aplicados como Lisp o Prolog, podríamos decir que cada lenguaje tiene su aplicación ideal dentro de la IA. Sin embargo, hay dos que destacan por su popularidad: **Python y R**.

Tanto Python como R son muy demandados en *big data*. La figura 1 muestra el nivel de popularidad basado en las búsquedas de Google de los términos “Python” y “R” seguidos de “data scientist”. Como vemos, ambos lenguajes se asocian con la ciencia de datos frecuentemente. A este respecto, ambos son opciones perfectamente válidas, con sus ventajas e inconvenientes (BBVA Open4U, 2016).



Figura 1. Comparativa de la popularidad relativa de los términos “python data scientist” y “R data scientist” en búsquedas de Google.

Python es mucho más comúnmente utilizado en el área del aprendizaje automático, como indica la figura 2. En gran medida, esto es el resultado de varios factores, entre los que se encuentran la facilidad de uso (sintaxis similar al inglés, sin necesidad de compilación, etc.) y la increíble asistencia que ofrece su comunidad (tanto en discusión de ideas como en ayuda para solventar problemas y acceso a herramientas en forma de librerías).



Figura 2. Comparativa de la popularidad relativa de los términos “Python machine learning” y “Machine learning with R” en búsquedas de Google.

R es un lenguaje formulado y pensado para el tratamiento estadístico y el manejo de datos. Así pues, es una elección popular en las matemáticas estadísticas y en ciertas áreas de la IA (como la ciencia de datos y el aprendizaje no supervisado). Sin embargo, carece del carácter generalista de Python, así como de su rico ecosistema.

En este curso nos centraremos en Python como lenguaje unificador dentro de la inteligencia artificial, sin que esta elección signifique que Python es innatamente superior a cualquier otro lenguaje. Como en otras áreas de la tecnología, el lenguaje de programación debe ajustarse a la tarea que se quiere completar. El caso es que, dado su rico ecosistema, Python es la elección más adecuada como común denominador dentro de la programación en IA. Como ejemplo, la inmensa mayoría de las librerías más utilizadas en la investigación en aprendizaje automático están escritas en Python o tienen interfaces para él: Scikit-learn (Pedregosa, Varoquaux, Gramfort, Michel y Thirion, 2011), TensorFlow (Abadi et al., 2016), Theano (Bergstra et al., 2011), Caffe (Jia et al., 2014) o Keras (Chollet, 2015).

1.3. Convenciones

En este manual nos ajustaremos a las siguientes convenciones en cuanto a formatos de texto:

Fuente normal en cursiva para links, ficheros, nombres y extensiones.

Fuente Courier New para código.

Fuente Courier New en negrita para comandos que el usuario debe escribir literalmente.

Fuente Courier New en cursiva para texto que debe ser reemplazado por el usuario en ejemplos y output de programas o consola de textos.

1.4. Librerías más populares y útiles

Python está en plena madurez como lenguaje, y prueba de ello es la cantidad de librerías desarrolladas que enriquecen su ecosistema. A continuación listamos varias de las más populares, tanto que casi se convierten en una extensión natural del lenguaje (en negrita las que cubriremos explícitamente en este curso):

- **NumPy:** Numerical Python, para computación numérica.
- **Pandas:** estructuras de datos.
- **Matplotlib:** gráficos y diagramas para visualización de datos.
- **Keras:** redes neuronales.
- **Nltk:** Natural Language ToolKit, procesamiento de lenguaje natural.
- **Beautiful Soup:** analizador de documentos HTML.
- **OpenCV:** visión artificial.
- **IPython y Jupyter:** computación explorativa e interactiva.
- SciPy: computación científica (optimización, estadística, álgebra lineal, etc.).
- **Scikit-learn:** aprendizaje automático general.
- Statsmodel: análisis estadístico.

En lo relativo a importar paquetes externos (que dotan a Python de funcionalidad añadida), para las librerías más populares utilizaremos nomenclaturas que se han adoptado como estándar:

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd
```

1.5. Instalación

El proceso de instalación de Python depende en gran medida del sistema operativo empleado. Aunque Python se puede instalar de forma independiente, se recomienda descargar e instalar Anaconda, un paquete de software que incluye Python y otras herramientas útiles en el desarrollo, algunas de las cuales veremos en el curso (**NumPy, Pandas, Matplotlib**).



Enlace de interés

Descarga e instrucciones para la instalación del paquete Anaconda.

<http://www.anaconda.com>

Para comprobar que se ha instalado correctamente, se puede abrir la línea de comandos en Windows **cmd** (o la consola en Linux y MacOS) e introducir el comando **python** (o **ipython** en MacOS) para iniciar el intérprete:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
```

Type "help", "copyright", "credits" or "license" for more information.

>>>

Para salir del intérprete, se puede introducir el comando **exit()** o bien pulsar **Ctrl + D**.

1.6. Python 2 vs. Python 3

Python nació en 1991. En 2008 fue publicada la tercera gran versión del lenguaje, Python 3. Hasta entonces, las versiones más populares eran las 2.x, y por ello todas las librerías estaban escritas para funcionar en esas versiones. La versión de Python 3 trajo consigo multitud de cambios, algunos de los cuales impedían la retrocompatibilidad del código escrito en Python 2.x. Por ese motivo, a pesar de la nueva versión, hasta hace bien poco Python 2.x continuó siendo la versión favorita de muchos programadores y desarrolladores de librerías.

Con el paso del tiempo, la mayoría de librerías han sido portadas a Python 3.x y hoy en día es la versión considerada como estándar. En este curso nos centraremos en Python 3.x. Concretamente, asumiremos la versión 3.6, aunque el código debería ser compatible con versiones más avanzadas.

1.7. Contenidos del curso

Este curso se centrará en proporcionar las bases para un conocimiento general del lenguaje Python, que luego se utilizará para explorar el entorno SciPy, un ecosistema gratuito en Python para las matemáticas, la ingeniería y la ciencia en general.

Tras exponer los fundamentos de Python, su sintaxis básica y las colecciones más comunes, se hará un repaso específico de las librerías más importantes del entorno SciPy: Jupyter Notebook (archivos interactivos de IPython), **NumPy** (colecciones multidimensionales y operaciones matemáticas), **Matplotlib** (visualización gráfica de datos), **Pandas** (estructuras de datos), **NLTK** (lenguaje natural) y **OpenCV** (visión artificial). Cada una de estas librerías es de uso común en análisis de datos, minería de datos y aprendizaje automático. Todos los ejemplos utilizados se enmarcan en la aplicación para su uso en estos dominios.

Para simplificar, el manual se referirá a **terminal** sin distinción de sistema operativo para indicar la línea de comandos (en Windows) o la terminal de comandos (en Linux y MacOS).



Capítulo 2

Python 101

A diferencia de otros lenguajes, como C, C++ o Java, Python es un **lenguaje interpretado**, lo que significa que no hace falta compilarlo (traducirlo a lenguaje de máquina) para ejecutarlo. Hay varias formas de ejecutar código en Python. En esta sección veremos las más comunes: el intérprete básico, **scripts** y Jupyter Notebook.

2.1. Intérprete básico

El **método** más básico de ejecutar código Python es utilizando el intérprete básico. Para ello, se introduce el comando **python** en la terminal. Esto abre el intérprete de Python, en el que se puede escribir, línea por línea, código Python.



Ejemplo

```
c:\Windows>python

Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914
64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> a = 5

>>> b = a * 2

>>> print(b)

10
```

2.2. Ejecución de scripts

El comando **python** también se puede utilizar para ejecutar **scripts**, que son archivos de texto que agrupan múltiples líneas de código. La utilización de scripts facilita la creación de bloques de código más complejos. Para ello, se pasa como parámetro el nombre del archivo **python script.py**.



Ejemplo

Archivo **test.py** contiene el siguiente texto:

```
a=5

b=a*2

print(b)
```

El siguiente comando produce el mismo output que en el ejemplo anterior, 10:

```
python test.py
```



Por convención, los scripts deben tener una extensión **.py** para indicar que son archivos ejecutables por el intérprete.

Como los *scripts* son simples archivos de texto, se puede utilizar cualquier editor para generarlos. Sin embargo, hay editores de texto especializados, además de IDE (entornos de desarrollo integrados), que pueden facilitar la escritura de código Python con características como resaltado de sintaxis, asistencia en el proceso de *debugging* (trazado del código en búsqueda de errores), etc. Visual Studio Code es una opción recomendable por su disponibilidad (gratis en todas las plataformas) y su funcionalidad, pero existen muchas más (Geany, Sublime, Notepad++, etc.). Como entornos de desarrollo, se encuentran Spyder, PyCharm, entre otros.

2.3. Jupyter Notebook

Jupyter es una iniciativa de código abierto para el desarrollo interactivo de programas en múltiples lenguajes. Jupyter Notebook es una aplicación web que permite crear y compartir documentos que contienen imágenes, texto y código ejecutable. Es una aplicación muy utilizada en la comunidad científica como herramienta de facilitación de la colaboración y el desarrollo iterativo en Python. La figura 3 muestra un ejemplo de un Jupyter Notebook en Python.

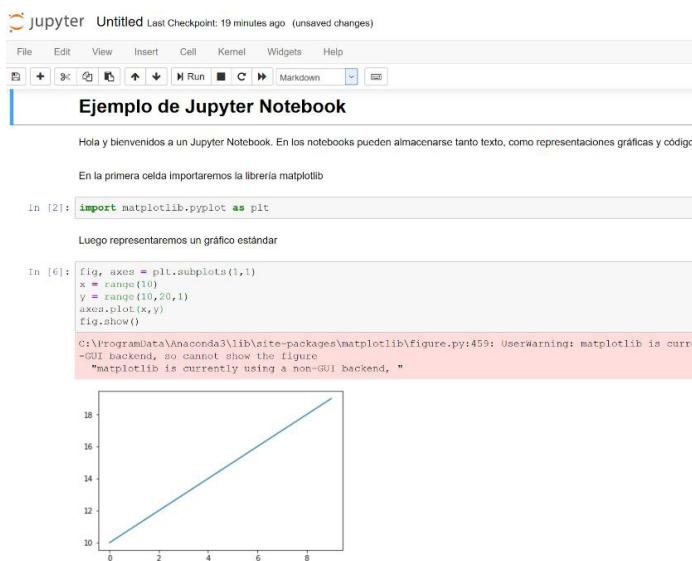


Figura 3. Ejemplo de Jupyter Notebook.

Un Jupyter Notebook consiste en una serie de celdas consecutivas. Cada celda puede tener uno de los siguientes tipos: *Code* (para código ejecutable), *Markdown* (para texto con formato), *Headings* (para encabezados; es una celda *markdown* con el carácter # al inicio) o *Raw NBConvert* (para texto sin formato). Se pueden crear celdas con el menú *Insert > New Cell* o con las teclas **alt + enter**. Todas las celdas son ejecutables (con el botón *Run* o con las teclas **shift + enter**). Las celdas *markdown* formatean el texto incluido al ejecutarse, mientras que las celdas *code* utilizan IPython, un intérprete de Python avanzado, para ejecutar el código.

Las sesiones escritas en el Notebook pueden grabarse en un archivo .pynb para acceso futuro, con el menú *File > Save and Checkpoint*. Esto permite compartir resultados fácilmente con otros colaboradores.



Enlace de interés

Aunque está fuera del temario de este curso, es posible compartir los notebooks manteniéndolos en un servidor remoto. De forma similar, Google ofrece una herramienta gratuita para mantener notebooks en su plataforma y así poder compartirlos con otros.

<https://colab.research.google.com/notebooks/welcome.ipynb>

2.4. Sintaxis básica

2.4.1. Variables

En programación, el uso de **variables** es fundamental. Una variable es una referencia a un valor; se puede entender como un cajón que contiene un elemento. En Python hay cinco **tipos de elementos** básicos: `float` (números reales), `int` (números enteros), `str` (texto), `bytes` (valores binarios) y `bool` (`True` o `False`). Además, existen dos tipos especiales: `None` (que indica ausencia de valor) y `NaN` (para indicar que una variable numérica no tiene un valor numérico, del inglés “*Not a Number*”). A diferencia de otros lenguajes, las variables no tienen un tipo estático, sino que el tipo es implícito y puede cambiar a lo largo del programa.

Para crear una variable, se escribe el nombre de la variable (debe ser único y empezar por un carácter alfabético) y se le asigna un valor con el signo `=`. Para asignar un valor textual (`string`), se pasa el valor entre comillado (tanto el entrecerrillado simple “” como el doble “” son aceptables). Una vez creadas, las variables se pueden utilizar en expresiones, del mismo modo en el que se usan en ecuaciones matemáticas. La Tabla 1 muestra un listado de operaciones binarias (entre dos elementos) en Python.

| Operación | Descripción |
|----------------------------------|--|
| <code>a + b</code> | suma a y b |
| <code>a - b</code> | resta a menos b |
| <code>a / b</code> | a dividido entre b |
| <code>a // b</code> | a dividido entre b (quitando decimales) |
| <code>a * b</code> | a multiplicado por b |
| <code>a ** b</code> | a elevado a b |
| <code>a % b</code> | devuelve el resto de la división a / b (modulus) |
| <code>a & b</code> | bool: True si a y b son True int: bitwise AND |
| <code>a b</code> | bool: True si a o b es True int: bitwise OR |
| <code>a ^ b</code> | bool: True si solo uno de los dos es True int: bitwise EXCLUSIVE OR |
| <code>a == b</code> | True si a y b son iguales |
| <code>a != b</code> | True si a y b son distintos |
| <code>a is b</code> | True si a y b son referencias al mismo objeto |
| <code>a is not b</code> | True si a y b son referencias a distintos objetos |
| <code>a <= b, a < b</code> | True si a es menor o igual que b (o solo menor) |
| <code>a >= b, a > b</code> | True si a es mayor o igual que b (o solo mayor) |

Tabla 1. Operaciones binarias en Python

```
str = 'hola mundo'

number1 = 42

number2 = 12

result = number1 - number2

print(result)

comb = str(number1) + str

print(comb)

number = 'forty two '


print(number1)
```

Programa 1. Ejemplo básico de programa Python.

```
30

42hola mundo

forty two
```

Salida 1. Salida del ejemplo de programa Python.

Como se puede ver en el ejemplo anterior, para aplicar operaciones (en este caso +, que entre objetos `string` se interpreta como concatenación) entre dos variables de distinto tipo, si la conversión no es obvia, se puede convertir (lo cual se denomina **casting**) una de ellas al tipo de la otra. En este caso, hemos convertido un valor `int` a `str` utilizando la expresión `str(number)`. Del mismo modo, podemos hacer casting a cualquier tipo utilizando `int()`, `float()`, `bool()`, `str()`.



La mayoría de los tipos básicos son interconvertibles. Sin embargo, hay *castings* que son inviables, como por ejemplo `int("a")`. En esos casos, el intérprete retornará un error de tipo `ValueError`, lo que indica que no se ha podido convertir el valor.

Aunque esto se verá con más detalle más adelante, se pueden crear objetos lista (*list*), que son una colección de elementos del mismo tipo. La forma más básica de crear una lista es con `[]`:

```
lista1 = [1, 5, -2]

lista2 = ['cefalopodo', 'mamifero', 'ave']
```

Cuando se quiere comprobar si dos variables son iguales, es preciso recalcar las diferencias entre comparar los elementos dentro de la variable y comparar las variables entre sí. Para comparar el valor de las variables, podemos usar `a == b`; si lo que queremos es comparar si dos variables hacen referencia al mismo objeto, debemos usar `a is b`.



Ejemplo

```
lista1 = [1,2,3]
lista2 = [1,2,3]
print(lista1 == lista2)
print(lista1 is lista2)
Output:
True
False
```

2.4.2. Comentarios

Los comentarios son partes del código que son ignoradas por el intérprete y, por lo tanto, ni se evalúan ni se ejecutan. Sirven para aportar información al lector sobre el funcionamiento del código, aunque también se utilizan durante el desarrollo para testear partes del código. En cualquier caso, son una pieza fundamental de cada programa, ya que aumentan la legibilidad del código si se usan adecuadamente. En Python, todos los caracteres escritos tras el símbolo `#` son considerados comentarios. Pueden aparecer al principio de la línea o tras una expresión.

2.4.3. Comparaciones condicionales

Si las variables son el primer elemento fundamental en los programas, las **comparaciones condicionales** son el segundo. A base de comparaciones condicionales se puede dirigir el flujo de ejecución para ejecutar una parte del programa en vez de otra. La sintaxis es la siguiente:

```
if <comparación>:
    <expresión1>
else:
    <expresión2>
```

En `<comparación>` se puede poner cualquier expresión que evalúe a `bool` (`True` o `False`). Si el valor es `True`, se ejecutará la `<expresión1>`; si es `False`, se ejecutará la `<expresión2>`. Ambas expresiones pueden contener múltiples frases.

```
a = 10
b = 5
if a < b:
    print("a is smaller")
else:
    printf("a is bigger or equal")
```

Programa 2. Uso de condicionales.

a is bigger or equal

Salida 2. Salida condicional, dado que a es > que b.



En Python no se utilizan paréntesis para estructurar el código, sino que se usa la indentación para crear bloques. En el caso de las comparaciones, ambas expresiones han de ir indentadas (al menos un espacio, pero todas las frases dentro de un bloque deben usar la misma indentación). Es comúnmente aceptado utilizar cuatro espacios o una tabulación como indentación de bloque. Si se necesita crear un bloque dentro de un bloque (por ejemplo, un `if` dentro de un `while` loop), las indentaciones son acumulables).

Se pueden concatenar comparaciones sustituyendo `else` por `elif <comparación2>`: para condicionales más complejos.

Aunque menos legible, hay una forma compacta de expresar comparaciones, el **operador ternario**. Se suele utilizar cuando las expresiones a ejecutar son muy sucintas, aunque se debe evitar si complica demasiado la legibilidad del código. La estructura del operador ternario equivalente al condicional anterior es:

`<expresión1> if <comparación> else <expresión2>`

2.4.4. Loops

Los **loops** permiten a los programas ejecutar código de forma iterativa o repetitiva. Hay dos tipos de expresiones que resultan en **loops**: `for` y `while`.

Los `for loops` se utilizan para iterar sobre una colección o **secuencia**, elemento a elemento. Tienen la siguiente forma:

```
for i in <secuencia>:
    <expresión>
```

`i` toma el valor de un elemento dentro de la `<secuencia>` en cada iteración.

Los `while loops` se utilizan para ejecutar una parte del código mientras una condición sea cierta. Tienen la siguiente forma:

```
while <condición>:
```

```
    <expresión>
```

Mientras la `<condición>` sea `True`, se ejecutará la `<expresión>`. Para evitar *loops* infinitos, la `<expresión>` debe modificar algún elemento en algún momento para que la `<comparación>` evalúe `False`.

Una **función** muy útil en *loops* (especialmente `for`) es `range`:

```
range(min, max, step)
```

`range` retorna una lista de elementos, desde `min` (inclusivo) hasta `max` (exclusivo), en incrementos definidos por `step`. El único argumento requerido es `max`. Si `min` no se incluye, se asume que es 0; `step` se asume 1. La llamada `range(1,10,2)` retorna la secuencia [1,3,5,7,9]. Esta secuencia se puede usar en el `for loop` para determinar el número de iteraciones.



Ejemplo

```
# for loop para imprimir todos los elementos de una lista

for n in range(10):

    print(n)

# while loop para imprimir los números pares menores de 10

a = 1

while a < 10:

    if a % 2 == 0:

        print(a)

    a = a + 1
```

En ocasiones es deseable saltar una iteración, es decir, parar la ejecución de la iteración actual y pasar a la siguiente. Esto se puede realizar con la palabra clave `continue`. También es posible salir de un *loop* forzadamente con la palabra clave `break`.



Ejemplo

Otra forma de escribir el `while loop` anterior que imprime los números pares menores de 10 es:

```
a = 1

while True:

    if a % 2 != 0:

        continue

    if a >= 10:

        break

    print(a)

    a++
```



Es importante apreciar que la diferencia entre ambos tipos de loops es simplemente semántica y que, por lo tanto, cualquier loop se puede expresar tanto con un `for` como con un `while` (aunque, dependiendo de la intención, uno será más legible y adecuado que el otro).

2.4.5. Funciones

Las **funciones** o **métodos** son formas de estructurar el código. Se utilizan para reutilizar código (por ejemplo, una función que se encargue de comprobar si un número es par) y también para separar lógicamente segmentos de código cuya funcionalidad está muy acotada (por ejemplo, un método que escanee una lista y reemplace ciertos elementos por 0).

Para definir una función antes de su utilización, se debe declarar:

```
def nombre(<lista de parámetros>):

    <expresión>

    return <lista de valores> # opcional
```

El nombre de la función puede ser cualquier nombre (siguiendo las mismas normas que para las variables). Cada función puede tener ninguno o más parámetros, indicados en <lista de parámetros> y separados por comas. Las funciones pueden acabar sin retornar un valor, pero pueden retornar uno o más valores, separados por comas.

```
def divisible_by(mylist, divisor):  
    divisible= []  
    for n in mylist:  
        divisible.append(n % divisor == 0)  
    return divisible, len(divisible)  
  
list1 = list(range(9))  
  
divs, count = evens(list1,2)  
  
print(count)  
  
print(divs)
```

Programa 3. Definición y uso de una función.

9

[True, False, True, False, True, False, True, False, True]

Salida 3. Ejemplo de uso de una función definida.

En Python, se puede asignar una palabra clave a cada argumento, para que los usuarios no tengan que recordar el orden o significado de cada argumento. Para ello, en el Programa 3 se modifica la llamada a la función divisible_by para indicar el nombre del parámetro: divisible_by(mylist=[1,2,3], divisor=2). Esa llamada es equivalente a divisible_by(divisor=2,mylist=[1,2,3]). También se pueden proponer valores por defecto: si el usuario no proporciona un argumento, la función asigna un valor fijo. Para ello, se cambia la definición de la función con el valor por defecto que se requiera:

```
def divisible_by(mylist, divisor=2):
```

Ahora ejecutar divisible_by([1,2,3]) es equivalente a divisible_by([1,2,3], 2).



Ejemplo

Se pueden utilizar parámetros por defecto, por ejemplo, para saber si el usuario ha proporcionado ciertos argumentos o no.

```
def f(a=None, b=None, c=None):  
  
    if a is not None:  
  
        print('Se ha proporcionado a')  
  
    if b is not None:  
  
        print('Se ha proporcionado b')  
  
    if c is not None:  
  
        print('Se ha proporcionado c')  
  
f(a=1,c="holá")
```

Programa 4. Parámetros por defecto en funciones Python.

Se ha proporcionado a

Se ha proporcionado c

Salida 4. Uso de parámetros por defecto.

En Python es perfectamente posible guardar una referencia a una función en una variable y luego ejecutar la función a través de dicha variable. En el ejemplo anterior, si asignamos la función `f` a la variable `b` (`b = f`) , obtenemos el mismo resultado con `f(a=1,c="holá")` que con `b(a=1,c="holá")` .

Si queremos aplicar una serie de funciones a cada objeto en una colección, es muy útil almacenar las funciones en una lista y aplicarlas sucesivamente. El Programa 5 muestra un ejemplo de esto, en el que se limpia una lista de strings para que cada elemento no tenga espacios en blanco a ambos lados (`str.strip`) y que empiece por mayúscula (`str.title`) .

```
def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result

strings = ['Valencia ', ' barcelona', ' bilbao ']
operations = [str.strip,str.title]
print(clean_strings(strings,operations))
```

Programa 5. Almacenamiento de referencias a funciones en una lista.

['Valencia', 'Barcelona', 'Bilbao']

Salida 5. Las strings originales han sido formateadas a través de las referencias a las funciones almacenadas en una lista.

También es posible definir funciones de forma anónima, sin nombre, a través de **funciones lambda**. Estas son útiles cuando la definición es sucinta y no se utiliza múltiples veces, por ejemplo, como argumento que se pasa a otra función. Suelen utilizarse para manipular un argumento de alguna forma. Las funciones lambda se escriben de la siguiente forma:

lambda <lista de argumentos> : <valor a retornar>

Ejemplo

Ordenación de las strings de una lista en función de su longitud:

```
strings = ["C","Carlos","Car"]
strings.sort(key=lambda x : len(x))
print(strings)
```

Programa 6. Ejemplo de utilización de función lambda.

['C', 'Car', 'Carlos']

Salida 6. Uso de una función lambda para ordenar strings.

2.5. Colecciones en Python

Todos los tipos básicos contienen un solo elemento. Es posible combinar varios elementos en un mismo objeto para formar una colección. En Python, hay varias clases de colecciones: tuplas, listas, secuencias, diccionarios y conjuntos.

2.5.1. Tuplas

Una **tupla** es una agrupación de elementos inmutable, con los elementos separados por comas y entre paréntesis: `(1, 2, 6, 10)` o `("Antonio", "Zorrilla", 29)`. Una tupla puede contener elementos de distinto tipo, pero, una vez creada, no es posible modificar, añadir o eliminar elementos, ni modificar los existentes. Para acceder a un elemento, se puede utilizar el operador `[]`. Por ejemplo, si `tuple1 = (1, 2, 3, 5)`, entonces `tuple1[1]` retorna el segundo elemento, en este caso, 2.

Las tuplas tienen dos métodos propios: `<tupla>.count(<elemento>)` retorna el número de ocurrencias del `<elemento>` en la tupla; `<tupla>.index(<elemento>)` retorna la posición del `<elemento>` en la tupla (si no existe, retorna un `ValueError`).

La sintaxis de Python permite desempaquetar una tupla asignando varias variables al mismo tiempo. Por ejemplo, `a, b, c = (1, 2, 3)` resulta en `a = 1`, `b = 2` y `c = 3`. También se puede desempaquetar parte de la tupla y poner el resto en una variable: `a, *otros = ("Carlos", "Gualberto", "Gherardo")` resulta en `a = "Carlos"` y `otros = ["Gualberto", "Gherardo"]`. Por cierto, internamente, este es el mecanismo que utiliza Python cuando una función retorna más de un valor.

2.5.2. Listas

Las **listas** son similares a las tuplas en cuanto a que son enumeraciones de elementos de cualquier tipo. A diferencia de las tuplas, las listas sí son mutables, es decir, es posible añadir, eliminar y modificar elementos. Para crear una lista, se enumeran los elementos entre corchetes, separados por comas: por ejemplo, `lista1 = ["No", "Si", 17, 0.121]`. También se utiliza el operador `[]` para acceder a los elementos de una lista y, como es mutable, se puede utilizar para modificar un elemento: `lista1[0] = None` resultaría en `[None, "Si", 17, 0.121]`.

Las listas también tienen los métodos `count()` e `index()` descritos para las tuplas. Hay tres funciones que permiten añadir elementos a las listas:

- `<lista>.append(<elemento>)` añade `<elemento>` al final de la lista.
- `<lista>.insert(<índice>, <elemento>)` añade `<elemento>` en la posición `<índice>`.
- `<lista>.extend(<secuencia>)` añade los elementos en la `<secuencia>` al final de la lista. Es funcionalmente equivalente a concatenar dos listas (`[1, 2, 3] + [4, 5, 6]`), pero evita la creación de una nueva lista.

Para eliminar elementos se puede utilizar `<lista>.remove(<elemento>)`, que elimina la primera ocurrencia de `<elemento>`, o `<lista>.pop([índices])`, que elimina y retorna los elementos en las posiciones indicadas en `[índices]`.

Otra función interesante es `<lista>.sort(reverse=False)`, que reordena la lista comparando los elementos entre sí. Sin embargo, como los elementos de una lista pueden tener tipos distintos, es importante asegurarse de que los elementos son comparables cuando se utiliza `sort`; de lo contrario, Python retornará un `TypeError`.

Una forma rápida de crear listas de secuencias es utilizando la función ya presentada `range()`. Por ejemplo, `list(range(-5, 6))` genera una lista del -5 al 5 en incrementos de 1 valor.

Python permite trabajar con elementos en listas de forma muy flexible gracias a lo que se conoce como **slicings** o cortes. Un *slicing* consiste en obtener una parte de una lista definiendo los límites (primer elemento y último) y el incremento, de forma muy similar a como se define `range()`. Los tres indicadores son opcionales y, si no se especifican, el intérprete usa el primer y último como límites y 1 como incremento. La forma general del *slicing* es `<lista>[<primero>:<último>:<incremento>]`. Se pueden indicar límites negativos, en cuyo caso cuenta desde el final de la lista (el límite -2 indica el tercer elemento desde el final de la lista).

La Figura 4 muestra la correspondencia entre índices positivos y negativos con una lista de letras.

| | | | | | | | | | | | |
|----|---|----|---|----|---|----|---|----|---|----|---|
| 0 | P | 1 | Y | 2 | T | 3 | H | 4 | O | 5 | N |
| -6 | | -5 | | -4 | | -3 | | -2 | | -1 | |

Figura 4. Correspondencia de índices positivos y negativos en una lista de letras `["P", "Y", "T", "H", "O", "N"]`.



Ejemplo

```
lista_completa = list(range(0,6,1)) # lista [0,1,2,3,4,5]
slice1 = lista_completa [:] # lista [0,1,2,3,4,5]
slice2 = lista_completa [1:3] # lista [1,2]
slice3 = lista_completa[:-1] # lista [0,1,2,3,4]
slice4 = lista_completa[::2] # lista [0,2,4]
```

2.5.3. Secuencias

Las **secuencias** son objetos iterables no materializados (no son listas, sino objetos). Aunque se pueden utilizar directamente en *loops*, no retornan una lista en sí (para ello se han de materializar con `list(<secuencia>)`). Ya hemos visto una forma de crear secuencias en Python: `range()`. Otra secuencia muy utilizada para iterar sobre una colección es `enumerate`, que retorna dos objetos iterables: el índice del elemento actual y su valor. Así, podemos iterar cada elemento de una lista de la siguiente forma:

```
for i, value in enumerate([10,20,30])
```

En cada *loop*, `i` tomará el valor del índice en cuestión (0, 1 y 2), mientras que `value` tomará el valor del elemento (10, 20 y 30).

`reversed(<secuencia>)` crea una secuencia invertida a partir de `<secuencia>` o colección.

Por último, la función `zip` crea una secuencia de tuplas a partir de dos listas, emparejando cada elemento. Toma la forma siguiente:

```
list(zip([e1,e2,e3],[o1,o2,o3])) retorna [(e1,o1),(e2,o2),(e3,o3)]
```

`zip` es muy útil cuando se desea iterar en un *loop* dos listas al mismo tiempo, como se muestra en el Programa 7.

```
nombres = ["Messi","Iniesta","Modric"]
```

```
edades = [31,34,34]
```

```
for (nombre,edad) in zip(nombres,edades):
```

```
    print("Nombre: " + nombre + ", edad: " + str(edad))
```

Programa 7. Uso de `zip` para iterar sobre dos listas simultáneamente.

Nombre: Messi, edad 31

Nombre: Iniesta, edad 34

Nombre: Modric, edad 34

Salida 7. En cada iteración, se accede a un elemento de cada lista.

Python permite hacer *unzip* mediante la sintaxis `*` (que internamente crea, como hemos visto, una tupla con la información restante):

```
lista1 = list(range(5))

lista2 = list(reversed(range(5)))

zipped = zip(lista1,lista2)

original1, original2 = zip(*zipped)

lista1 == original1 # True

lista2 == original2 # True
```

2.5.4. Diccionarios

Los **diccionarios** son colecciones de elementos que incluyen parejas de claves (referencias únicas) y valores. Cualquier objeto inmutable puede ser clave; cualquier objeto puede ser utilizado como valor, tanto elementos básicos como diccionarios u otras colecciones (diccionarios anidados). La forma de crear un diccionario básico es:

```
dict1 = {<clave1>:<valor1>, <clave2>:<valor2>, ...}
```

Con el operador `[]` se accede a los valores del diccionario, pasando el índice deseado (clave). Por ejemplo, si `dict1 = {"Nombre": "Carlos", "Apellido": "Fernandez"}`, entonces `dict1["-Nombre"]` retorna "Carlos". También se puede acceder para modificar valores: `dict1["Nombre"] = "Leo"` cambia el nombre de la primera entrada.

Si al acceder o modificar un diccionario la clave no está presente, Python la genera implícitamente: `dict1["Edad"] = 34` añade la pareja "Edad":34 al diccionario `dict1`. Para comprobar si una clave está presente en el diccionario, se puede usar el operador `in`. Por ejemplo, "Nombre" `in dict1` retorna `True`, pero "Profesion" `in dict1` retorna `False`.

Se pueden eliminar claves de un diccionario de dos formas: del `dict1[key]` elimina la clave `key` del diccionario; `dict1.pop(key)` hace lo mismo.

Para iterar un diccionario, se puede hacer enumerando sus claves (`for key in dict1.keys()`) o directamente con `for key in dict1`, enumerando sus valores (`for value in dict1.values()`) o incluso ambos.

```
for key,value in zip(dict1.keys(),dict1.values())
```

Aquí hemos creado dos secuencias separadas, con las claves y los valores, y las hemos unido con `zip`. Luego, en cada iteración, tenemos la clave (`key`) y el valor (`value`) por separado.

Nota: también se pueden iterar claves y valores utilizando `<diccionario>.items()`.

Además de poder crear diccionarios explícitamente con el par de caracteres `{ }`, Python permite transformar secuencias en diccionarios fácilmente, utilizando `zip`:

```
nombres = ["Messi", "Iniesta", "Modric"]
edades = [31, 34, 34]
diccionario = dict( zip(nombres,edades) ) # { "Messi": 31, "Iniesta": 34,
"Modric": 34 }
```

2.5.5. Sets (conjuntos)

La última colección que examinaremos son los `sets` o conjuntos, que son una forma específica de diccionarios sin valores, solo claves. De esta forma, cada elemento en un set está representado un máximo de una vez (como el concepto de conjunto en matemáticas). Cualquier objeto inmutable puede ser un elemento en un set (tipos básicos o tuplas, por ejemplo).

Para crear un `set`, se utiliza la función `set(<[elementos]>)`, pasando un listado de elementos. Por ejemplo, `set([1,1,2,3,5])` retorna el set `{1,2,3,5}`.

Como con los conjuntos matemáticos, hay varias operaciones básicas que se pueden hacer entre sets:

- Unión (signo `|`): retorna todos los elementos de ambos sets.
- Intersección (signo `&`): retorna solo los elementos presentes en ambos sets.
- Diferencia (signo `-`): retorna los elementos del primer set que no se encuentran en el segundo.
- Diferencia simétrica (signo `^`): retorna solo los elementos contenidos en un solo set y no en el otro.

También hay ciertas funciones que facilitan la comparación entre sets:

- `<set1>.issubset(<set2>)` retorna True si `<set1>` es un subconjunto de `<set2>`.
- `<set1>.issuperset(<set2>)` retorna True si `<set1>` es un superconjunto de `<set2>`.
- `<set1>.isdisjoint(<set2>)` retorna True si `<set1>` y `<set2>` son sets disjuntos (es decir, si su intersección es nula).

2.5.6. Python comprehension

Uno de los casos más citados como ejemplo del poder de la sintaxis básica de Python son las *list comprehensions* (aunque también se pueden usar en diccionarios). Se utilizan para construir listas, diccionarios o conjuntos a través de expresiones más sucintas que el equivalente `for loop`. La forma general de *list comprehension* es la siguiente:

```
[<expresión1> if <condition> else <expresión2> for value in <collection>]
```

Esta expresión genera una lista en la que cada elemento está formado evaluando la `<expresión1>` (que normalmente incluye algún tipo de transformación de `value`). Como vemos, puede también incluirse un condicional para usar `<expresión1>` o `<expresión2>` en cada iteración.

Dictionary / set comprehension es similar, pero en vez de usar el par de caracteres `[]` se utiliza el par `{}` para generar la colección.



Ejemplo

```
# Usando for loop básico

numbers = list(range(-10,10))

values = []

for x in numbers:

    if(x > 0):

        values .append(x*2)

    else:

        values .append(x/2)

# Usando list comprehension

values = [x*2 if x>0 else x/2 for x in range(-10,10)]

# usando dict comprehension para intercambiar claves y valores

compra= {"Bananas":1.5, "Peras":2.5, "Uvas":1.2}

rev_dict = {value:key for key,value in compra.items()} # {1.5:"Bananas",
2.5:"Peras", 1.2:"Uvas"}
```

Aunque en ocasiones dificulta la legibilidad, es posible anidar *comprehensions* concatenándolas. La expresión [elemento for item in compra.items() for elemento, precio in item] es equivalente al for loop anidado siguiente:

```
precios = []  
  
for item in compra.items():  
  
    for elemento,precio in item:  
  
        precios.append(precio)
```



Capítulo 3

Colecciones: NumPy

NumPy es una librería externa para Python. Su nombre deriva de “**Numerical Python**” y facilita la computación numérica en Python. Para utilizarla se ha de importar el módulo `numpy` con la expresión `import numpy`. La siguiente expresión importa el módulo `numpy` y lo hace accesible a través del alias `np`:

```
import numpy as np
```

NumPy está escrito en lenguaje C, por lo que es muy eficiente y rápido de usar. Es el estándar de facto en cuanto a computación numérica, y muchas otras librerías lo tratan como tal.

NumPy se utiliza frecuentemente para crear colecciones multidimensionales y aplicar funciones matemáticas estándares a los elementos en una colección.

3.1. Objeto básico en NumPy: ndarray

En el corazón de **NumPy** está el objeto `ndarray`. Una `ndarray` es una colección multidimensional de datos del mismo tipo, aunque se pueden usar `ndarrays` para colecciones de `bool` o `string`, son útiles y muy eficientes cuando se usan para grabar y manipular valores numéricos (`float` e `int`).



Cada ndarray solo puede contener elementos de un solo tipo (int, float, string, bool, etc.). Si hay que crear una ndarray que contenga tipos diferentes, Python trata de convertir los tipos a uno solo, por ejemplo:

- `np.array([1,"2"])` se convierte en `array(["1","2"])`
- `np.array([True,0])` se convierte en `array([1,0])`

Para crear ndarrays, **NumPy** ofrece varias formas:

- `np.array(<secuencia>)` crea una ndarray a partir de una lista o secuencia de Python. Si la lista o secuencia es anidada, se creará una dimensión por nivel. Ejemplo: `a = np.array([[1,2], [3,4,5]])` crea la ndarray bidimensional `array ([1,2],[3,4,5])`
- `np.zeros(<dimensiones>)`, `np.ones(<dimensiones>)` y `np.empty(<dimensiones>)` generan una ndarray con valores 0, 1 o sin determinar, respectivamente. El valor `dimensiones` se pasa como una tupla. Ejemplo: `np.zeros((2,2))` genera `array([0,0], [0,0])`. Con `np.empty`, los elementos asignados inicialmente son indeterminados, es decir, pueden tomar cualquier valor.
- `np.arange(min,max,step)` crea una ndarray de forma similar a la secuencia `range`.
- `np.full(<tamaño>,<valor>)` crea una ndarray de tamaño definido por la tupla `<tamaño>` con todos los valores inicializados a `<valor>`.

Algunos de los métodos más interesantes en cualquier objeto ndarray son los siguientes:

- `sort()` ordena los elementos de la array.
- `argmax()` retorna el índice del elemento más alto.
- `argmin()` retorna el índice del elemento más bajo.



Enlace de interés

Existen muchos métodos aplicables a cada ndarray. Aunque aquí solo se cubrirán algunos, se puede consultar el resto en el siguiente enlace:

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>

3.1.1. Dimensiones

Cada ndarray contiene metainformación sobre sus dimensiones y el tipo de sus elementos. Así, `array1.ndim` retorna el número de dimensiones, `array1.shape` retorna una tupla con las dimensiones y `array1.dtype` retorna el tipo de sus elementos.

NumPy permite forzar una ndarray a unas dimensiones específicas. A través del método `reshape (x, y)` de cada ndarray, -x determina el número de elementos en la primera dimensión, e y determina el número de elementos en la segunda. Por ejemplo, la `arr1 = np.array([1, 2, 3, 4])` puede cambiar a una 2D con `arr1.reshape(2, 2)`, resultando en `array([[1, 2], [3, 4]])`.

Un ejemplo en el que `reshape` es muy conveniente es cuando se desea generar una matriz con valores consecutivos. `np.arange(9).reshape(3, 3)` retorna la matriz:

```
[ [0, 1, 2],
  [3, 4, 5],
  [6, 7, 8] ]
```

Como opuesto a `reshape`, `flatten()` se utiliza para allanar una ndarray multidimensional a una de una sola dimensión.

3.1.2. Manipulación de ndarrays

NumPy permite juntar dos arrays, bien de lado a lado o bien una encima de la otra, con las funciones `np.hstack(a1, a2)` o `np.vstack(a1, a2)` respectivamente (del inglés *horizontal stack* y *vertical stack*).

También es posible dividir una array en múltiples. `np.array_split(array1, divisiones)` divide `array1` en tantas partes como se define en `divisiones`. Se puede pasar el argumento `axis` para determinar si se debe dividir por filas (`axis=0`) o columnas (`axis=1`). `divisiones` puede ser una tupla o lista (de tamaño máximo igual a las dimensiones de `array1`), en cuyo caso se puede especificar dónde cortar en cada dimensión.

```
a = np.arange(16).reshape(4, 4)

dividida = np.array_split(a, (3, 3) )

for i, d in enumerate(dividida):

    print(i)

    print(d)
```

Programa 8. Manipulación de las dimensiones de una ndarray.

```
0
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
1
[]
2
[[12 13 14 15]]
```

Salida 8. ndarray dividida con el método split.

3.1.3. Índices y *slicing*

Tanto los índices como *slicing* en ndarrays funcionan de manera homóloga a como lo hacen en listas Python, con la diferencia de que, cuando se solicita una slice de una ndarray, se retorna una referencia a la parte afectada de la ndarray, y no una copia. Esto significa que si se modifica algún elemento de la slice, el cambio se verá reflejado en la ndarray original.

NumPy permite mucha más flexibilidad para realizar y trabajar con slices. Por ejemplo, se puede asignar un único valor a los elementos de la slice con `array1[:3] = 0` (asigna 0 a los tres primeros elementos). También se pueden hacer slices condicionales, es decir, retornar solo los elementos que cumplen una condición. Para ello, se especifica una condición (o condiciones encadenadas con operadores lógicos &, | o ~) entre corchetes.

Ejemplo

```
array1 = np.arange(-3,4) # [-3,-2,-1,0,1,2,3]
array1[array1 < 0] # [-3,-2,-1]
array1[array1 % 2 == 0] = 10 # [-3, 10, -1, 10, 1, 10, 3]
array1[array1 == 10] = 0 # [-3, 0, -1, 0, 1, 0, 3]
```

Para más flexibilidad, se puede utilizar una lista como índice a una ndarray, en cuyo caso se retornan los elementos con los índices especificados en la lista (si la lista es 2D, el primer elemento determina la fila y el segundo la columna). Por ejemplo, `array[[1,2]]` retorna el segundo y tercer elementos.

Cuando se trabaja con ndarrays multidimensionales se puede acceder a elementos a través de índices separados por comas. En la ndarray bidimensional `array1`, `array1[0][1] == array[0,1]`.

3.2. Funciones matemáticas sobre colecciones

Uno de los aspectos que más valor tiene de **NumPy** es la facilidad de aplicar operaciones o funciones matemáticas a un conjunto de valores de una vez, sin necesidad de iterar sobre cada uno de los valores con un *loop*. Así, se puede aplicar cualquier **operador binario** (+, -, /, *, %, etc.) a toda una ndarray de la misma forma que con cualquier variable:

```
array = np.arange(5) # array(0,1,2,3,4)
array = array * 10 # array(0,10,20,30,40)
```

También se pueden utilizar operadores entre dos ndarrays, lo que resulta en una ndarray producto de la operación entre cada uno de los elementos:

```
array1 = np.array([1,1,1,1])
array2 = np.array([5,5,5,5])
result = array1 + array2 # array(6,6,6,6)
```



Enlace de interés

Si se aplican operadores binarios a ndarrays con dimensiones distintas, Python aplica la técnica de **broadcasting**, que, en su forma más simple, resulta en virtualmente incrementar el tamaño de la ndarray más pequeña hasta igualar el de la otra, para así poder aplicar la operación correctamente. Esto es lo que sucede en el caso básico de aplicar una operación entre un escalar y una ndarray (por ejemplo, `np.arange(10) * 2`).

El valor de la porción incrementada es normalmente una copia del elemento ya existente en la ndarray, aunque en situaciones más complejas se determina siguiendo una serie de reglas. Para más información acerca de las reglas de broadcasting se puede visitar el siguiente enlace:

<https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html>

Además de poder aplicar operaciones Python a ndarrays, **NumPy** ofrece una serie de **operadores unarios** (que se aplican a un solo operando) y **operadores binarios**. La Tabla 2 lista algunos de los más útiles.

Nota: para utilizar los operadores, se debe hacer a través del alias np (o el que se haya decidido al importar **NumPy**). Por ejemplo, `np.abs(np.array([1,-2,3]))` retorna `array(1,2,3)`.

| Tipo | Operación | Descripción |
|---------------|-------------------------------|--|
| Unario | <code>abs</code> | Valor absoluto de cada elemento. |
| | <code>sqrt</code> | Raíz cuadrada de cada elemento. |
| | <code>exp</code> | e^x , siendo x cada elemento. |
| | <code>log, log10, log2</code> | Logaritmos en distintas bases de cada elemento. |
| | <code>sign</code> | Retorna el signo de cada elemento (-1 para negativo, 0 o 1 para positivo). |
| | <code>ceil</code> | Redondea cada elemento por arriba. |
| | <code>floor</code> | Redondea cada elemento por abajo. |
| | <code>isnan</code> | Retorna si cada elemento es NaN. |

| Tipo | Operación | Descripción |
|----------------|---|--|
| | <code>cos, sin, tan</code> | Operaciones trigonométricas en cada elemento. |
| | <code>arccos, arcsin, arctan</code> | Inversas de operaciones trigonométricas en cada elemento. |
| Binario | <code>add</code> | Suma de dos arrays. |
| | <code>subtract</code> | Resta de dos arrays. |
| | <code>multiply</code> | Multiplicación de dos arrays. |
| | <code>divide</code> | División de dos arrays. |
| | <code>maximum, minimum</code> | Retorna el valor máximo / mínimo de cada pareja de elementos. |
| | <code>equal, not_equal</code> | Retorna la comparación (igual o no igual) de cada pareja de elementos. |
| | <code>greater, greater_equal, less, less_equal</code> | Retorna la comparación (>, >=, <, <= respectivamente) de cada pareja de elementos. |

Tabla 2. Listado de los operadores binarios y unarios más comunes dentro de **NumPy**

3.3. Filtrado de datos

Una función muy útil en el análisis de datos, por la versatilidad que da para filtrar y modificar información, es `np.where`. En su forma más básica, retorna una `ndarray` resultado de aplicar una condición elemento a elemento sobre una `ndarray`, escogiendo un valor `a` o el valor `b` en función del resultado de la comparación. Toma la siguiente forma:

```
np.where(<condición>, <elemento_condición_afirmativa> , <elemento_ condición_ negativa>)
```

Por ejemplo, se puede utilizar `np.where` para crear una lista en la que los elementos positivos aparecen con un 1 y los demás con 0 de la siguiente forma:

```
resultado = np.where(array1 < 0, 0, 1)
```

`np.where` permite también utilizar `ndarrays` como elementos sustitutivos, de forma que, en vez de sustituir los valores con condicional afirmativo por un único valor, se escoge un elemento de una `ndarray`. Por ejemplo, siguiendo el ejemplo anterior, se pueden sustituir todos los elementos negativos por 0, pero dejar los positivos como están:

```
resultado = np.where(array1 < 0, 0, array1)
```

Si se proporciona solo el valor condicional, **NumPy** retorna los índices para los que la condición es afirmativa. Por ejemplo, para conocer la posición de los elementos negativos de `array = np.array([-1,1,-2,2])`, se puede hacer a través de `np.where(array < 0)`, que retorna `array([0,2])`.

También se puede emplear `np.where` para limpieza básica de datos, por ejemplo, para eliminar ocurrencias de `NaN` en los datos. Si tenemos `array = np.array([1,2,np.nan])`, aplicando `np.where(np.isnan(array), 0, array)` obtenemos `array([1,2,0])`.

3.4. Exploración y estadística descriptiva

Una parte fundamental del analista de datos es entender las características de la información que maneja. Así, la fase preliminar de exploración ayuda al profesional a comprender la naturaleza de los datos que se tienen entre manos. En este sentido, la estadística descriptiva ofrece una visión general de los datos en conjunto como el valor medio, cómo de diferentes son los valores, mínimos y máximos, etc.

En la Tabla 3 pueden observarse algunas de las funciones de estadística descriptiva dentro de **NumPy**. Todas son accesibles a través del alias **NumPy**, pasando como argumento la ndarray que se desea explorar. Por ejemplo, np.mean(arr1) retorna el valor medio de la array arr1.

| Función | Descripción |
|----------------------------|--|
| sum(arr1) | Suma de todos los elementos de arr1. |
| mean(arr1) | Media aritmética de los elementos en arr1. |
| std(arr1) | Desviación estándar de los elementos en arr1. |
| cumsum(arr1) | Retorna array con la suma acumulada de cada elemento con todos los anteriores. |
| cumprod(arr1) | Retorna array con el producto acumulado de cada elemento con todos los anteriores. |
| min(arr1), max(arr1) | Mínimo y máximo de arr1. |
| any(arr1) | En array de tipo bool, retorna True si algún elemento es True. |
| all(arr1) | En array de tipo bool, retorna True si todos los elementos son True (o >0 en valores numéricos). |
| unique(arr1) | Retorna una array con valores únicos (similar a set). |
| in1d(arr1, arr2) | Retorna array con bool indicando si cada elemento de arr1 está en arr2. |
| union1d(arr1, arr2) | Retorna la unión de ambas arrays. |
| intersect1d(arr1, arr2) | Retorna la intersección de ambas arrays. |

Tabla 3. Funciones en **NumPy** de estadística descriptiva

Si se trabaja con ndarrays multidimensionales, la mayoría de funciones permiten pasar un argumento para indicar qué eje de la ndarray considerar. De esta forma se puede obtener la suma de todos los elementos, columna por columna, de una array de la siguiente forma:

```
array1 = np.array([ [1,2,3], [10,20,30] ])

np.sum(array1, axis=0) # array([11,22,33])
```

En este caso, axis acepta el eje (o dimensión): 0 para columnas, 1 para filas, y así sucesivamente si hay más dimensiones.

3.5. Álgebra lineal

NumPy, como buena librería de arrays, dispone de funciones y métodos para facilitar el trabajo en álgebra lineal, desde operaciones con matrices hasta resolución de sistemas de ecuaciones.

Para multiplicar dos matrices expresadas como ndarrays se puede utilizar `np.dot(matriz1, matriz2)`, o bien directamente `matriz1 @ matriz2` (en Python 3.5+). También se puede obtener el mismo resultado con el método propio de una array, `matriz1.dot(matriz2)`.

Nota: `matriz1 * matriz2` resulta en una multiplicación elemento a elemento y no en el producto de dos matrices. Para que la multiplicación tenga sentido, las dimensiones de las matrices deben ser compatibles (si el tamaño de `matriz1` es $A \times B$, el tamaño de `matriz2` debe ser $B \times C$, es decir, el número de columnas de la primera matriz debe coincidir con el número de filas de la segunda).

Otras operaciones convenientes para ndarrays que representan matrices son:

- Calcular la traspuesta con `matriz1.T` o `matriz1.transpose()`.
- Obtener los elementos en la diagonal con `matriz1.diagonal()`.
- Construir una matriz con elementos específicos en su diagonal (pero 0 en el resto): `np.diag(array)`, con `array` siendo unidimensional.

NumPy incluye un submódulo llamado `numpy.linalg` dedicado al álgebra lineal, con funcionalidad dedicada a la factorización de matrices, cálculo de determinantes e inversas, etc. La Tabla 4 describe algunas de las funciones más importantes.

| Función | Descripción |
|---------------------------------|--|
| <code>dot(mat1, mat2)</code> | Retorna el producto escalar entre dos arrays. Si son matrices 2D, es equivalente a la multiplicación de ambas. |
| <code>matmul(mat1, mat2)</code> | Retorna el producto entre dos matrices. |
| <code>trace(mat1, mat2)</code> | Suma de las diagonales de ambas matrices. |
| <code>det(mat1)</code> | Retorna el determinante de la matriz <code>mat1</code> . |
| <code>eig(mat1)</code> | Computa los autovalores y autovectores de la matriz cuadrada <code>mat1</code> . |
| <code>inv(mat1)</code> | Retorna la inversa de la matriz <code>mat1</code> . |
| <code>qr(mat1)</code> | Computa la factorización QR de <code>mat1</code> . |
| <code>solve(A, b)</code> | Resuelve el sistema lineal de ecuaciones $Ax = b$ para x , cuando A es una matriz cuadrada. |

Tabla 4. Lista de las funciones más importantes en `numpy.linalg`



Enlace de interés

Para más información sobre las funciones dentro de `numpy.linalg` se puede acceder a la documentación oficial en el siguiente enlace:

<https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.linalg.html>

3.6. Valores aleatorios

Python contiene un módulo denominado `random` que se puede utilizar para generar números aleatorios. Una vez importado con `import random`, se pueden generar números aleatorios con `random.random()` para fracciones de 0 a 1, `random.randint(min, max)` para números enteros dentro de un rango o `random.uniform(low, high)` para fracciones dentro de un rango. También ofrece la posibilidad de elegir un valor de entre una colección al azar con `random.choice(<colección>)`, donde colección puede ser también una `string`.

Aunque el módulo `random` puede ser útil en casos simples, tiende a ser relativamente lento. **NumPy** ofrece una serie de funciones dentro del `namespace np.random` con mejor rendimiento. Entre otras se encuentran las siguientes:

- `np.random.rand()` retorna un número aleatorio entre 0 y 1.
- `np.random.randint(min, max)` retorna un número entero entre `min` (inclusive) y `max` (exclusive).
- `np.random.randn(d0, d1...)` retorna una `ndarray` con elementos aleatorios de una distribución normal (media 0 y varianza 1) de dimensiones especificadas por `d0, d1...`

Para obtener números a partir de distintas distribuciones estadísticas, **NumPy** ofrece los siguientes métodos:

- `np.random.binomial(n, p)` genera números de una distribución binomial.
- `np.random.uniform(low, high)` genera números a partir de una distribución uniforme (con igual probabilidad para todos).
- `np.random.poisson(lambda, size)` genera números de una distribución Poisson.



Enlace de interés

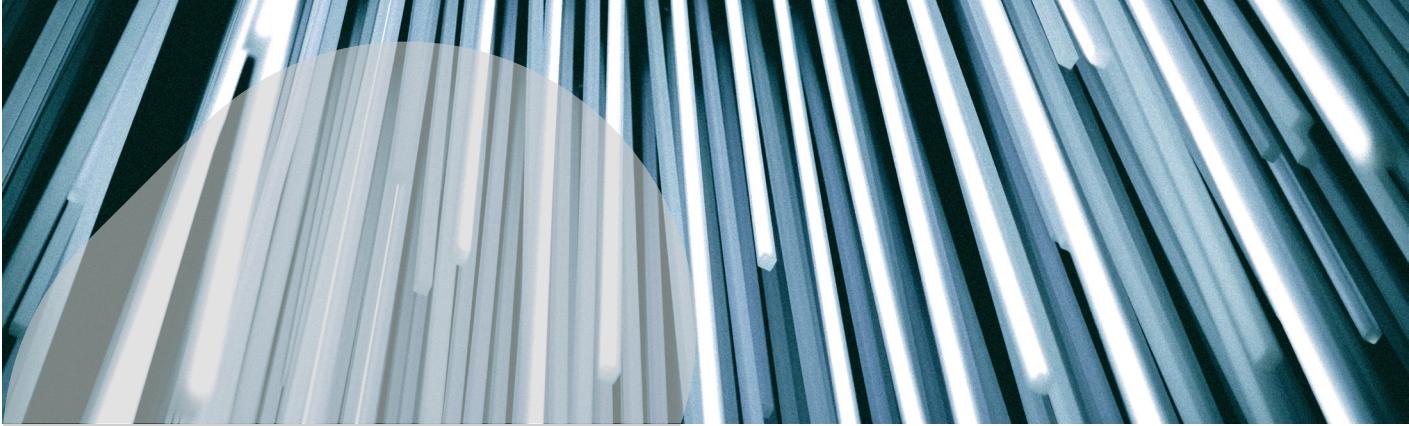
Para más información sobre números aleatorios en **NumPy** y otras distribuciones se puede visitar el siguiente enlace:

<https://docs.scipy.org/doc/numpy/reference/routines.random.html>



Por supuesto, como máquinas deterministas que son, los ordenadores no pueden generar números realmente aleatorios. Los algoritmos utilizan ecuaciones para generar números pseudoaleatorios, pero siempre a partir de un valor dado, que se denomina semilla. Esto quiere decir que una secuencia de números pseudoaleatorios a partir de una semilla es siempre la misma, aunque para el observador tenga un patrón aleatorio. Si se escoge otra semilla, la secuencia será distinta. A menudo, esto sirve en simulaciones científicas para asegurarse de que las condiciones de un experimento son idénticas.

Para iniciar una secuencia pseudoaleatoria con una semilla determinada, en **NumPy** se utiliza el método `np.random.seed(semilla)`.



Capítulo 4

Estructuras de datos: Pandas

NumPy está diseñado específicamente para trabajar de forma eficiente con colecciones de datos numéricos multidimensionales. Sin embargo, cuando los datos son de varios tipos (`string`, `int`, `float`), **NumPy** puede llegar a ser incómodo e ineficiente. Desgraciadamente, tanto en aprendizaje automático como en análisis de datos, la situación más frecuente es tener datos heterogéneos, como la colección de datos sobre reseñas cinematográficas *MovieLens 20M Dataset*, que incluye información como el nombre de la película, el género (`string`), el año de publicación (`int`) y la votación del usuario (`float`).



Enlace de interés

Para más información sobre MovieLens 20M Dataset, incluido un enlace de descarga, se puede acceder a la página de MovieLens:

<https://grouplens.org/datasets/movielens/>

Pandas es una librería de código abierto para Python que contiene estructuras de datos y herramientas para trabajar naturalmente con datos heterogéneos (o **datos tabulares**, como son llamados comúnmente). Es el estándar de facto para datos tabulares en Python y, junto con **NumPy**, **Matplotlib** y **Scikit-learn**, forma parte de la colección científica **SciPy**.

La forma más habitual para utilizar Pandas es hacerlo a través del alias `pd`:

```
import pandas as pd
```

Si se desea ser más específico se puede importar exclusivamente una parte de **Pandas**, por ejemplo:

```
from pandas import Series, DataFrame
```

Importar directamente partes de una librería evita tener que repetir el alias múltiples veces. Así, se puede utilizar `Series`, en vez de `pd.Series`. `Series` y `DataFrame` son las dos estructuras de datos básicas en **Pandas**. Las veremos con detalle en las próximas secciones.

4.1. Series

Las `Series` son estructuras unidimensionales similares a las `ndarray` de **NumPy**, en las que cada elemento posee también un índice único. La forma más sencilla de crear una serie es con el constructor `pd.Series(<lista>, index=<lista de índices>)`. El argumento `index` es opcional y, si no se pasa, **Pandas** asume índices de 0 a $n-1$ (donde n es el tamaño de la `<lista>` de elementos).

```
serie = pd.Series(['Barcelona', 'Madrid', 'Valencia', 'Sevilla'],  
index=['a', 'b', 'c', 'd'])  
  
series.name = 'Al menos dos equipos en primera'  
  
series.index.name = 'Id'  
  
print(series)
```

Programa 9. Ejemplo de utilización del objeto `Series` de **Pandas**.

```
Id  
  
a Barcelona  
  
b Madrid  
  
c Valencia  
  
d Sevilla  
  
Name: Al menos dos equipos en primera, dtype: object
```

Salida 9. Imprimir un objeto `Series` muestra el nombre del índice, el nombre de la serie, el tipo de objeto de cada elemento y los elementos incluidos (junto a sus índices).

Como se puede ver en el ejemplo anterior, tanto la serie (`serie.name`) como la columna de índices (`serie.index.name`) pueden nombrarse para facilitar su introspección.

El acceso a los elementos de una serie se puede hacer a través del índice explícito o en listas, por orden. En el ejemplo, `serie['d'] == serie[3]`. En general se puede trabajar con Series de la misma forma que con ndarrays:

- Listado de índices para acceso a múltiples elementos: tanto a través del índice explícito (`serie[['a', 'c']]`) como el ordinal (`serie[[0, 2]]`).
- Índice condicional para filtrar elementos: `serie[serie > 10]` retorna una serie solo con los elementos mayores de 10.
- Slicing: `serie[:-1]` retorna una serie con todos los elementos menos el último.
- Funciones aplicadas a todos los elementos: `np.sqrt(serie)` retorna una serie en la que cada elemento es la raíz cuadrada del original.



Como cualquier otra colección, se puede iterar sobre los valores de una serie con un simple `for loop`.

```
for index, value in enumerate(serie):
```

En cada iteración, `index` toma el valor del índice del elemento, y `value` es el elemento en sí. Si solo interesa iterar sobre los valores, se puede hacer directamente sin `enumerate` en la forma simplificada: `for value in serie`.

Como las Series son estructuras con índice explícito, se pueden entender como simples diccionarios Python en que la clave es el índice y el valor es el elemento. Como tales, se pueden aplicar operaciones de diccionarios:

- Averiguar si un índice está presente: `'b' in serie`
- Convertir un diccionario en una serie: `serie = pd.Series({ 'Carlos' : 100, 'Marcos' : 98 })`. Dado que un set es un diccionario de claves sin valores, también se puede convertir un set en una serie de la misma forma.

Pandas hace posible la unión de dos Series con el operador `+`. Cuando se unen dos Series, el resultado es una serie cuyos elementos se determinan en función de los índices:

- Los elementos cuyo índice está en ambas series se suman.
- Los elementos cuyo índice solo aparece en una serie se añaden con el valor NaN.



Ejemplo

```
serie1 = pd.Series([10,20,30,40],index=range(4) )  
serie2 = pd.Series([1,2,3],index=range(3) )  
serieComb = (serie1 + serie2)  
  
# serieComb contiene lo siguiente  
  
0      11.0  
1      22.0  
2      33.0  
3      NaN  
  
dtype: float64
```

El operador – funciona de forma similar, pero restando los valores de los elementos cuyo índice aparece en ambas series.

Con ambos operadores es posible acabar con valores perdidos (*missing values*) si hay elementos únicos en cualquiera de las series. Para estos casos, y para el caso general de filtrar los datos, **Pandas** ofrece funciones para detectar dichos valores. `pd.isnull(<serie>)` retorna una serie cuyos valores corresponden a `True` si el valor de la `<serie>` es nulo, y `False` si no lo es. Un caso práctico es sustituir los valores perdidos de una serie con un valor fijo. Para ello se utiliza el resultado de `pd.isnull()` para filtrar elementos de una serie y asignar un valor, como se muestra en el Programa 10.

```
serie1 = pd.Series([1,2,3,np.NaN])  
serie1[ pd.isnull(serie1) ] = 0  
  
print(serie1)
```

Programa 10. Utilización de `pd.isnull` para filtrar y reemplazar valores nulos en una serie.

```
0      1.0  
1      2.0  
2      3.0  
3      0.0  
  
dtype: float64
```

Salida 10.

La función `pd.notnull(<serie>)` es equivalente pero contraria (retorna `True` cuando el elemento no es nulo).

4.2. DataFrame

Para trabajar con datos tabulares (filas y columnas), **Pandas** incluye la versátil estructura DataFrame. Un DataFrame o *frame* se puede entender como una colección de Series (columnas), todas compartiendo un listado de índices únicos. La forma más común de crear un *frame* es con un diccionario en el que cada clave se asocia a un listado de elementos de igual longitud.

```
diccionario = { "nombre" : ["Marisa", "Laura", "Manuel"], "edad" : [34, 29, 11] }

frame = pd.DataFrame(diccionario)

   nombre    edad
0  Marina     24
1   Laura     29
2  Manuel     11
```

Como podemos observar, la clave se utiliza como nombre de cada columna (serie), y cada elemento se asocia a una fila en función del índice. Cada columna puede tener un tipo de elemento (en este caso, '`nombre`' tiene tipo `string` y '`edad`' el tipo `int`).

Como ocurre con Series, en las que se puede especificar el listado a utilizar como índice, en DataFrames se puede pasar un listado de nombres para columnas con el argumento `columns` en el constructor. Esto sirve también para determinar el orden de las columnas en el *frame*.

```
pd.DataFrame(<diccionario>, columns=<listado de columnas>)
```

Implícitamente, **Pandas** automáticamente detecta si una clave en el diccionario aparece en el listado de columnas; cuando el nombre de la columna no aparece como clave, **Pandas** crea una nueva columna en el *frame* con valores NaN para todas las filas.



Ejemplo

```
diccionario = {"nombre" : ["Marisa", "Laura", "Manuel"], "edad" : [34, 29, 11] }

frame = pd.DataFrame(diccionario, columns = [ 'Nacionalidad', 'nombre',
'edad', 'profesion' ] )

# frame contiene lo siguiente

   Nacionalidad    nombre    edad    profesión
0        NaN      Marina     24        NaN
1        NaN       Laura     29        NaN
2        NaN      Manuel     11        NaN
```

Una vez creado el `frame`, se puede acceder a cada una de sus columnas por separado utilizando como índice el nombre de la columna, bien con corchetes (`<frame>['columna']`) o directamente (`<frame>.columna`). En el ejemplo anterior se puede acceder a la columna `edad` con `frame['edad']` o `frame.edad`.

Con frecuencia, los `DataFrames` pueden tener miles o millones de entradas, por lo que a veces es conveniente, para explorar los datos, observar solo los primeros casos, con `<frame>.head()`.

Añadir datos a un `frame` es sencillo y análogo a como se hace en diccionarios o `ndarrays`. Para cambiar los elementos de una columna se asigna directamente con `<frame>['columna'] = <valor>`. Si la columna no existe, se creará una nueva. Si ya existe, los elementos se sustituyen con `<valor>` (si la columna ya existe, también se puede asignar directamente con `<frame>.columna`). Nótese que el valor puede ser un elemento simple (numérico, texto, etc.), pero también puede ser una `Series`, de modo que se puede inicializar el valor de cada fila en la tabla.

Para eliminar columnas se puede utilizar el comando del `<frame>['columna']` o utilizar el método `<objeto>.drop(<índices>)`, válido tanto para `Series` como para `DataFrame`.

```
d_and_d_characters = {'Name' : ['bundenth','theorin','barlok'], 'Strength' : [10,12,19], 'Wisdom' : [20,13,6]}

character_data = pd.DataFrame(d_and_d_characters)

print('Original data')

print(character_data)

vitality_data = pd.Series([11,10,14])

character_data['vitality'] = vitality_data

character_data['alive'] = True

print('Modified data')

print(character_data)

del character_data['alive']

print('Final data')

print(character_data)
```

Programa 11. Ejemplo de creación y manipulación de un `DataFrame`.

Original data

Name Strength Wisdom

0 bundenth 10 20

```

1 theorin 12 13
2 barlok 19 6

Modified data

    Name Strength Wisdom vitality alive
0 bundenth 10 20 11 True
1 theorin 12 13 10 True
2 barlok 19 6 14 True

Final data

    Name Strength Wisdom vitality
0 bundenth 10 20 11
1 theorin 12 13 10
2 barlok 19 6 14
  
```

Salida 11. Creación y manipulación de un DataFrame.

`<frame>.values` retorna los datos del *frame* en formato lista; cada elemento es una fila de la tabla original, cada elemento de la cual toma el valor de la fila en la columna correspondiente. En el ejemplo de los personajes de Dungeons and Dragons, `character_data.values[0]` retorna `[‘bundenth’, 10, 20, 11]`. Acceder a los `values` puede ser útil para iterar sobre todos los elementos de una fila.

Como si de una matriz se tratara, se puede acceder al DataFrame inverso (cambiar filas por columnas y viceversa) con `<frame>.T`.

4.3. Trabajo con Series y DataFrame

En esta sección se verán funcionalidades en Pandas que se pueden aplicar tanto a objetos Series como DataFrame como búsqueda de elementos, eliminación de datos, indexación, *slicing* y reindexación.

4.3.1. Búsqueda

El método `<objeto>.isin(<lista>)` retorna una serie o *frame* (dependiendo del tipo de `<objeto>`) máscara, de las mismas dimensiones que `<objeto>` pero con el valor de cada elemento siendo `True` si el valor original se encuentra dentro de `<lista>` y `False` si no está presente. De forma más específica, se puede obtener una máscara con los elementos que son nulos (`<objeto>.isnull()`) o `NaN` (`<objeto>.isna()`).

También se pueden formular *queries* para determinar si una fila (búsqueda por índice) o columna (búsqueda por columna) existe, del mismo modo que se hace con los diccionarios o las listas. Por ejemplo, “Carlos” in obj.index retorna True si hay una entrada con índice Carlos en obj. Para buscar columnas, se puede hacer a través del listado <frame>.columns, por ejemplo, “age” in frame.columns.

4.3.2. Indexación y slicing

Como mencionamos en la sección de Series, tanto indexación como *slicing* en **Pandas** funcionan de forma análoga a como lo hacen en **NumPy**. En el caso de Series, es posible indexar por posición (como en **NumPy** o colecciones Python) o por referencia (utilizando el **índice semántico** de la serie). Por ejemplo, en la serie = pd.Series([3,43,12],index=['first','second','third']), se puede acceder al primer valor con serie[0] y con serie['first'].

En el caso de DataFrame, solo es posible indexar por referencia (por nombre de la columna), retornando una serie con los valores de la columna especificada.

Tanto en DataFrame como en Series es posible que haya lugar a ambigüedades cuando se indexa con números enteros; objeto[0] puede ser interpretado como índice posicional (acceder al primer valor) o índice semántico (acceder al valor con índice 0). Por ello, **Pandas** siempre interpreta un número entero como índice posicional. Para desambiguar estas situaciones, **Pandas** ofrece dos métodos: <objeto>.loc[X] accede al elemento con índice semántico X, mientras que <objeto>.iloc[X] accede al elemento con índice posicional x. El Programa 12 muestra la diferencia de resultados con ambos métodos. Nótese que a través del método iloc es posible indexar posicionalmente un DataFrame. En el caso de DataFrame, es posible definir a qué eje iloc o loc hacen referencia mediante el argumento axis (0 para filas, 1 para columnas). Así, frame.iloc(axis=1)[0] retorna los datos de la primera columna de frame.

```
frame = pd.DataFrame({'Name' : ['Carlos', 'Pedro'], "Age" : [34,22]}, index=[1,0])

print('Frame')

print(frame)

print('Primera fila')

print(frame.iloc[0])

print('Elemento con index 0')

print(frame.loc[0])
```

Programa 12. Indexación posicional y semántica a través de loc e iloc.

Frame

```
Name      Age
1   Carlos    24
0   Pedro     22
```

Primera fila

Name Carlos

Age 34

Name: 1, dtype: object

Elemento con index 0

Name Pedro

Age 22

Name: 0, dtype: object

Salida 12. El output es diferente para loc e iloc cuando el índice semántico y el posicional no coinciden.

De la misma forma que se puede indexar por posición o referencia, **Pandas** permite hacer *slicing* con índices ordinales y semánticos. El *slicing* con índices posicionales funciona de manera idéntica al resto de colecciones en Python. *Slicing* con índices semánticos es similar, pero con la peculiaridad de que se incluyen ambos extremos del rango (al contrario que con los posicionales, en los que el elemento de la derecha no se incluye). La forma general de *slicing* con índices semánticos es la siguiente:

<objeto>[<index1>:<index2>:<step>]

Slicing con índices semánticos funciona directamente con Series pero no con DataFrame. Para DataFrame se puede hacer *slicing* posicional o semántico empleando los métodos loc e iloc, pero, en vez de utilizar un solo índice como en el Programa 12, se define una slice. Por ejemplo, en el frame definido en el Programa 12, los siguientes comandos obtienen los siguientes resultados definidos:

```
frame.iloc(axis=1)[0] # primera columna (nombres)

frame.loc(axis=1)["Name"] # primera columna (nombres)

frame.iloc(axis=0)[1:] # segunda fila ("pedro", 22)

frame.loc(axis=0)[1:] # filas a partir del índice semántico 1 ("carlos" y "pedro")
```

`loc` e `iloc` son útiles para filtrar columnas si se desea trabajar solo con un subconjunto de ellas. En general, se utiliza `<frame>.iloc[X, Y]` para obtener los elementos con filas X y columnas Y (ambos pueden ser listas). `<frame>.loc[X, Y]` funciona de manera análoga pero con índices semánticos. Por ejemplo, en el frame de D&D del Programa 11, se pueden obtener solo las columnas Strength y Vitality con `character_data.loc[:, ["Strength", "Vitality"]]`, o todas las columnas desde Name hasta Wisdom con `character_data.loc[:, "Name": "Wisdom"]`.

Como en **NumPy**, es posible la indexación condicional, muy útil para la filtración o detección de datos. Por ejemplo, se pueden obtener las filas de los elementos mayores de edad con el comando `frame[frame["Age"] > 18]`. En general, al utilizar índices condicionales, Python retorna los elementos para los que la condición es True.

Por último, es posible utilizar índices posicionales y semánticos para eliminar datos de una serie o `frame` a través del método ya mencionado `<objeto>.drop(<índices>)`.

4.3.3. Ordenación de Series y DataFrame

Hay diversas formas de ordenar series y `frames`:

- `<objeto>.sort_index()` ordena el objeto alfabéticamente según el índice de cada elemento. Argumentos opcionales `axis=0` o `1` y `ascending=True` o `False`.
- `<serie>.sort_value()` es similar a `sort_index()`, pero se utiliza el valor de cada elemento para ordenar.
- `<frame>.sort_values(by=<columna>)` ordena las filas con respecto al valor en la `<columna>` de cada elemento. Acepta `ascending`.

Además de reordenar las estructuras, **Pandas** puede ordenar categóricamente objetos asignando un valor ordinal a cada elemento. Esto es de utilidad cuando se desea hacer un ranking de valores. El método `<objeto>.rank()` retorna un objeto (`frame` o `serie`) en el que cada elemento es la posición que este ocupa en relación con el resto de su columna.



Ejemplo

Con la matriz aleatoria `rand_matrix` siguiente:

```
rand_matrix = np.random.randint(6, size=(2,3))

frame = pd.DataFrame(rand_matrix , columns=list('ABC'))

      A    B    C
0      4    4    1
1      1    2    5
```

>>>

Ejemplo

Un *ranking* queda de la siguiente forma:

```
frame.rank()

      A      B      C
0    2.0   2.0   1.0
1    1.0   1.0   2.9
```

Como en todas las funciones de ordenación, `rank()` acepta el argumento opcional `ascending`.

4.4. Operaciones con Series y DataFrame

Los objetos en **Pandas** (series y frames) incluyen métodos aritméticos para sumar, restar, dividir, multiplicar, elevar a potencia, etc. entre objetos del mismo tipo. Este es un listado de algunos de los métodos aritméticos más comunes y su símbolo equivalente:

```
<objeto1>.add(<objeto2>) == <objeto1> + <objeto2>
<objeto1>.sub(<objeto2>) == <objeto1> - <objeto2>
<objeto1>.mul(<objeto2>) == <objeto1> * <objeto2>
<objeto1>.div(<objeto2>) == <objeto1> / <objeto2>
```

Internamente, **Pandas** compara los elementos de los objetos de la operación y, si está presente en ambos, el resultado es la aplicación del operando. Si el elemento solo aparece en uno, se añade un nuevo elemento al resultado con valor `NaN`. Si se prefiere, se puede cambiar el comportamiento por defecto de rellenar con `NaN`. Para ello, se pasa el argumento `fill_value=<valor>` a cualquiera de los métodos. Por ejemplo, `serie1.add(serie2, fill_value=0)` rellenará las casillas donde no se puede aplicar la suma con 0.



Nótese que, al aplicar una operación a un *frame* o una serie, la operación debe estar definida y, por lo tanto, ser válida para el tipo de elemento contenido en el objeto. Es decir, para sumar dos *Series*, el operador `+` debe estar definido para el tipo de elemento en la *Series*. Si el tipo de objeto es `string`, nótese que el signo `+` está definido como concatenación y es válido, pero el signo `-` no lo es, de modo que Python retornará un error `unsupported operand type(s) for -: 'str' and 'str'`.

A los operadores aritméticos se les suman otros operadores, como los operadores lógicos:

```
<objeto1> or <objeto2>
<objeto1> not <objeto2>
```

Técnicamente es posible aplicar operaciones aritméticas entre Series y DataFrame, para lo que **Pandas** alinea basándose en el índice para ejercer la operación. Un uso común es, por ejemplo, ver la diferencia de una fila en un DataFrame respecto al resto. Se puede ver un ejemplo de ello en el Programa 13. Como se vio en el capítulo sobre **NumPy**, este tipo de operaciones entre elementos bidimensionales y unidimensionales es posible gracias al *broadcasting*.

```
rand_matrix = np.random.randint(10, size=(3, 4))

print(rand_matrix)

df      = pd.DataFrame(rand_matrix           , columns=list('ABCD'))
print(df - df.iloc[0])
```

Programa 13. Ejemplo de operación aritmética entre un frame y una serie.

```
[ [5 4 2 8]
  [5 1 1 3]
  [6 3 3 7] ]
```

| | A | B | C | D |
|---|---|----|----|----|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | -3 | -1 | -5 |
| 2 | 1 | -1 | 1 | -1 |

Salida 13. El resultado final muestra la diferencia de todas las filas con la primera.



Como **Pandas** está construido sobre **NumPy**, todas las operaciones unarias y binarias descritas anteriormente y recogidas en la Tabla 2 son aplicables a series y frames.

Otra herramienta muy versátil es la aplicación de funciones lambda a series y frames mediante el método `<objeto>.apply(<función_lambda>)`. Por defecto, la **función lambda** será aplicada columna por columna, aunque se puede pasar el argumento axis para definir el objetivo (0 para filas, 1 para columnas). El Programa 14 muestra un ejemplo de la utilización de funciones lambda para calcular la diferencia entre el valor máximo y mínimo en cada columna.

```
rand_matrix = np.arange(12).reshape(3, 4)

frame = pd.DataFrame(rand_matrix , columns=list('ABCD'))

print(frame)

frame.apply(lambda x : x.max() - x.min())

print(frame)
```

Programa 14. Uso de funciones lambda para averiguar la diferencia entre valores en cada columna de un *frame*.

| | A | B | C | D |
|---|---|---|----|----|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 | 7 |
| 2 | 8 | 9 | 10 | 11 |

| | A | B | C | D |
|---|---|---|---|---|
| A | 8 | | | |
| B | 8 | | | |
| C | 8 | | | |
| D | 8 | | | |

dtype: int64

Salida 14. La función lambda es aplicada a cada columna del *frame*.

4.5. Estadística con Series y DataFrame

Para entender mejor los datos que se poseen, es muy útil la exploración. **Pandas** ofrece varios métodos de estadística sumaria para Series y DataFrame que ayudan a ver los datos en su conjunto.

En Series y DataFrames:

- <objeto>.sum() retorna el valor acumulado de todos los valores (en series) o por columna (en frames). Acepta el argumento axis para determinar si la suma se hace por fila o columna.
- <objeto>.mean() retorna lo mismo que sum pero con la media de valores.
- <objeto>.cumsum() retorna una suma acumulativa por elementos. Cada elemento es el valor acumulado de los elementos precedentes.
- <objeto>.idxmin() y <objeto>.idxmax() retorna el índice semántico (fila) del elemento menor o mayor de cada columna.

La Tabla 5 contiene algunos de los métodos de estadística sumaria más importantes para series y *frames*.

| Método | Descripción |
|----------------|--|
| count | Número de valores no NaN |
| describe | Conjunto de estadísticas sumarias |
| min, max | Valores mínimo y máximo |
| argmin, argmax | Índices posicionales del valor mínimo y máximo |
| idxmin, idxmax | Índices semánticos del valor mínimo y máximo |
| sum | Suma de los elementos |
| mean | Media de los elementos |
| median | Mediana de los elementos |
| mad | Desviación absoluta media del valor medio |
| var | Varianza de los elementos |
| std | Desviación estándar de los elementos |
| cumsum | Suma acumulada de los elementos |
| diff | Diferencia aritmética de los elementos |

Tabla 5. Métodos más comunes de estadística descriptiva para Series y DataFrame

Además de la estadística descriptiva, **Pandas** reúne métodos para entender la relación matemática entre dos conjuntos de valores.

- `<serie1>.corr(serie2)`: correlación entre las dos series `[-1, 1]`.
- `<frame>.corr()`: matriz de correlación en el frame. Mide la variación entre columnas.
- `<frame1>.corrwith(<frame2>)`: grado de correlación entre frames. Acepta el argumento `axis` para determinar la base de la comparación (0 filas, 1 columnas).



Capítulo 5

Visualización: Matplotlib

En capítulos anteriores hemos visto diversas formas de extraer información acerca de los datos, como métodos de estadística descriptiva en **NumPy** y en **Pandas**. Estas funciones nos proporcionan información muy valiosa para entender la naturaleza de los datos, como la necesidad de transformaciones o la detección de *outliers*.

Gran parte del tiempo del analista de datos se emplea en esta fase exploratoria, y sin duda una de las técnicas más utilizadas es la visualización de datos. Representarlos visualmente puede complementar su exploración, pero también ayudar a generar ideas para modelos en fases avanzadas. Al otro lado del espectro, representar los resultados de los modelos implementados de forma gráfica ayuda a cuantificar las mejoras alcanzadas y a presentar nuestras ideas de forma concisa.

En Python, la librería más comúnmente utilizada para la visualización de datos y la representación gráfica en general es **Matplotlib**. **Matplotlib** permite la generación de diagramas y gráficos de alta calidad en múltiples formatos, entre los que se encuentran los gráficos vectoriales (ideales para no perder resolución al escalar). Para utilizarlo:

```
import matplotlib.pyplot as plt
```

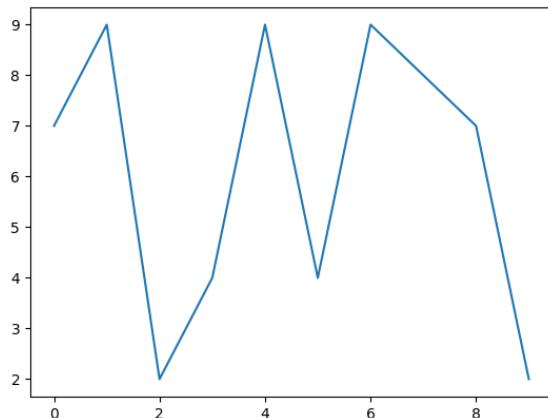
Si se está trabajando dentro de un Jupyter Notebook, se puede activar con el comando `%matplotlib notebook`.

5.1. Matplotlib 101

La forma más directa de crear un gráfico es a través del método `plot(<data>)`, donde `<data>` es el conjunto de datos que se quiere representar. El Programa 15 muestra un ejemplo.

```
import numpy as np
import matplotlib.pyplot as plt
data = np.random.randint(10, size=(10))
data # array([7, 9, 2, 4, 9, 4, 9, 8, 7, 2])
plt.plot(data) # creación del gráfico
plt.show() # comando que muestra los gráficos creados hasta el momento
```

Programa 15. Representación de datos de forma gráfica con `plt.plot()`.



Salida 15. Gráfico que representa los datos generados por **NumPy**.

Como se puede observar en el Programa 15, `plt.plot()` recibe los datos que se desean representar. En su forma general, la función acepta dos listas, una para cada eje (`X` e `Y`), en el que cada elemento es un punto en el gráfico. En este ejemplo, como solo se proporciona una lista de valores, **Matplotlib** asume que son los que representan la coordenada `Y`, y que la `X` son valores de 0 a 9 (o hasta el tamaño que indique la lista). En general, la siguiente expresión genera un gráfico utilizando los datos de `x_data` e `y_data`:

```
plt.plot(x_data, y_data)
```

Como se indica en el Programa 15, Python no presenta el gráfico directamente al usuario, sino que este se mantiene en la memoria hasta que se ejecuta el comando `plt.show()`. Esto se puede utilizar para trabajar con varios gráficos al mismo tiempo y presentarlos todos a la vez con un solo comando.

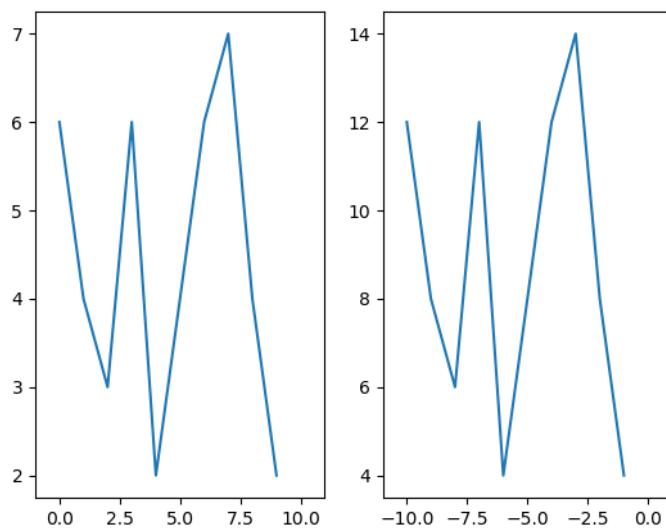
Los gráficos en **Matplotlib** residen dentro del objeto `Figure`. Este objeto se utiliza para representar un gráfico, pero también permite la subdivisión para múltiples gráficos. Para obtener una referencia al objeto general, se puede ejecutar `fig = plt.figure()`, que retorna dicha referencia. Una vez obtenida la referencia, se puede subdividir la figura con `fig.add_subplot(X, Y, 1)`, que la divide en X filas e Y columnas. No obstante, **Matplotlib** presenta un método directo para generar figuras y subfiguras:

```
fig, axes = plt.subplots(X, Y)
```

`subplots` retorna una referencia a la figura en general (`fig`) y una `ndarray` bidimensional (X filas e Y columnas) con referencias a cada subfigura (`axes`). Una vez obtenida una referencia para cada subfigura, se puede proceder a dibujar en cada una con el método `plot()`. Nótese que, al ser una `ndarray`, `axes` puede indexarse fácilmente por coordenadas; así, se accede a la subfigura en la primera fila y segunda columna a través de `axes[0, 1]`. El Programa 16 muestra un ejemplo de cómo trabajar con varias subfiguras.

```
x_data1 = np.arange(0,10)
x_data2 = np.arange(-10,0)
y_data1 = np.random.randint(10, size=(10))
y_data2 = y_data1 * 2
fig, axes = plt.subplots(1,2)
#preparar subfigura1
axes[0].set_xlim([-1,11])
axes[0].plot(x_data1,y_data1)
#preparar subfigura2
axes[1].set_xlim([-11,1])
axes[1].plot(x_data2,y_data2)
#mostrar ambas figuras
fig.show()
```

Programa 16. Ejemplo de creación de gráficos en distintas subfiguras.



Salida 16. Cada gráfico se muestra en una de las dos subfiguras creadas.



Enlace de interés

Matplotlib tiene una colección extensa de métodos para configurar cada gráfico, como el rango de valores de cada eje, el espacio entre figuras, los colores a utilizar, etc. A continuación se ofrece un enlace a la API para más información:

https://matplotlib.org/api/pyplot_api.html

Sin embargo, es una documentación muy amplia, por lo que se recomienda visitar antes la página de ejemplos de figuras para descubrir cómo realizar gráficos de distintos tipos:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.figure.html

5.2. Múltiples gráficos

Tanto el método general `plt.plot()` como el que reside en figuras y subfiguras aceptan varios argumentos que afectan a cómo se representa la información. Entre los más utilizados se encuentran `plt.plot(x, y, style, label, linewidth=1, alpha=1)`:

- `style: string` que define el estilo con el que dibujar los datos. Sigue la forma '`[color] [marcador] [línea]`', en la que cada opción se define como en la tabla 6. Un ejemplo es '`ro-`' para marcadores circulares, rojos con líneas continuas.
- `label:` utilizado en la leyenda como referencia al gráfico actual.
- `linewidth:` ancho de la línea al dibujar.
- `alpha:` nivel de opacidad. Se puede utilizar para dar transparencia a la figura.

| Color | Descripción | Marcador | Descripción |
|-------|---------------------|----------|-----------------------|
| b | blue | . | point marker |
| g | green | , | pixel marker |
| r | red | o | circle marker |
| c | cyan | v | triangle_down marker |
| m | magenta | ^ | triangle_up marker |
| y | yellow | < | triangle_left marker |
| k | black | > | triangle_right marker |
| w | white | 1 | tri_down marker |
| | | 2 | tri_up marker |
| Línea | Descripción | 3 | tri_left marker |
| - | solid line style | 4 | tri_right marker |
| -- | dashed line style | s | square marker |
| -. | dash-dot line style | p | pentagon marker |
| : | dotted line style | * | star marker |
| | | h | hexagon1 marker |
| | | H | hexagon2 marker |
| | | + | plus marker |
| | | x | x marker |
| | | D | diamond marker |
| | | d | thin_diamond marker |
| | | | vline marker |
| | | _ | hline marker |

Tabla 6. Información sobre cómo formar la string `style` para determinar el estilo con `plot()`.



Enlace de interés

Para más información sobre el método `plot()` y sus argumentos:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

En ocasiones es importante representar más de un gráfico dentro de una misma figura, por ejemplo, para comparar dos tendencias o múltiples alternativas. Para ello, podemos hacer más de una llamada a la función `plot()` **en la misma figura o subfigura**. Para ayudar a la visualización de elementos distintos, cada llamada puede tener diferentes parámetros de `style` y `label`. Cuando se tiene más de un gráfico en una figura, tradicionalmente se añade una leyenda que nombra a cada uno de ellos. Para mostrar la leyenda, se utiliza el método `legend(loc='best')`, bien el general a través de `plt.legend` o bien en la figura o subfigura.

El Programa 17 muestra un ejemplo de cómo representar dos gráficos en una figura.

```
x_data = np.arange(10)

y_data1 = np.random.randint(10, size=(10))

y_data2 = np.random.randint(10, size=(10))

fig, axes = plt.subplots(1) #solo solicitamos una figura

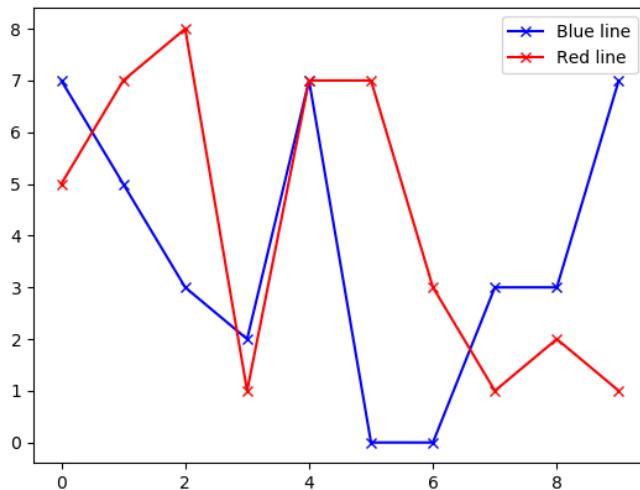
axes.plot(x_data,y_data1,'bx-',label='Blue line')

axes.plot(x_data,y_data2,'rx-',label='Red line')

axes.legend(loc='best')

fig.show()
```

Programa 17. Dos gráficos dentro de la misma figura.



Salida 17. Los dos gráficos residen dentro de la misma figura y se incluye una leyenda.

5.3. Decoraciones y anotaciones

Además de poder definir el estilo del gráfico en la función `plot()`, **Matplotlib** ofrece un conjunto de métodos **decoradores** que ayudan a modificar el aspecto de la figura. Desde cambiar los límites representados en el gráfico hasta añadir títulos, hay una gran colección de métodos, tanto a través de `plt` como en la clase `Axes`. Siguiendo el estilo de programación orientada a objetos, puede decorarse cada subfigura a través del objeto `axes` (como hasta ahora). Algunos de los decoradores más habituales son:

- `axes.set_xticks(<lista>)` y `set_yticks(<lista>)` definen explícitamente qué números aparecen escritos en los ejes X e Y.
- `axes.set_title('titulo')` establece el título general de la subfigura.
- `axes.set_xlabel('label')` y `set_ylabel('label')` establecen el nombre para cada eje.



Enlace de interés

Más información sobre decoradores en el siguiente enlace:

https://matplotlib.org/api/axes_api.html#plotting

A través de anotaciones se puede añadir texto y otros dibujos a los gráficos. Es posible añadir texto directamente en las coordenadas (x, y) con el método `axes.text(x, y, 'texto')`. Para facilitar las anotaciones, **Matplotlib** ofrece el método `axes.annotate()`, que permite no solo añadir texto, sino incluir flechas. La forma general del método es la siguiente:

```
annotate(texto, xy, xytext, xycoords, arrowprops)
```

`texto` es la `string` que queremos representar; `xy` son las coordenadas (tupla) en las que deseamos poner la anotación; `xytext` son las coordenadas (tupla) donde se escribe el texto; `xycoords` define el sistema de coordenadas en el cual `xy` está expresado; `arrowprops` es un diccionario que define el tipo y estilo de flecha utilizado.

La siguiente anotación produce el gráfico de más abajo:

```
ax.annotate('resaltar', xy=(2, 1), xytext=(3, 1.5), arrowprops = dict(facecolor='black', shrink=0.05))
```

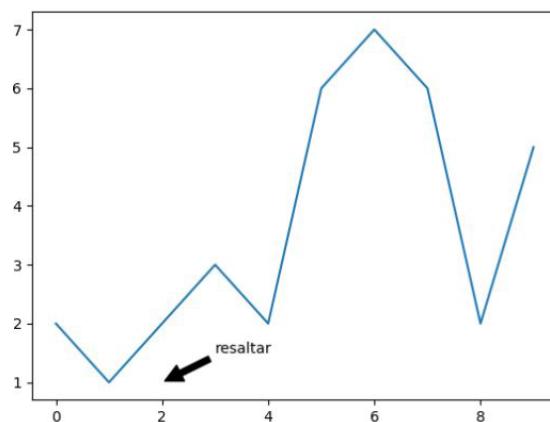


Figura 5. Ejemplo de anotación en un gráfico.



Enlace de interés

Más información y ejemplos de anotaciones en el siguiente enlace:

<https://matplotlib.org/users/annotations.html>

5.4. Tipos de gráficos

Matplotlib incluye funciones específicas para representar la información de distintas formas, como `scatter plot`, histograma, diagrama de pastel, gráfico con errores, etc. En este curso utilizaremos la interfaz de **Pandas** (véase epígrafe 5.6) para elegir el tipo de gráfico con el fin de abstraernos de los detalles de bajo nivel. No obstante, el siguiente enlace ofrece información sobre cómo utilizar **Matplotlib** para este fin:

https://matplotlib.org/api/axes_api.html#plotting

5.5. Grabación de gráficos a archivo

Una vez producida la figura (individual o múltiple), `plt.savefig(<filename>)` la guarda en el disco duro. `<filename>` indica al mismo tiempo el directorio de destino, el nombre del archivo a utilizar y la extensión.

Matplotlib puede deducir el formato de la imagen con la extensión de `<filename>`, pero se puede forzar con el parámetro `format`, que acepta una `string` con el nombre de la extensión (**Matplotlib** da soporte a una multitud de formatos, entre los que destacan jpg, png para imágenes de píxeles y svg, eps y pdf para imágenes vectoriales). La forma general de la función es, con parámetros por defecto:

```
plt.savefig(<filename>, dpi=None, facecolor='w', edgecolor='w', format=None, bbox_inches=None)
```

- `<filename>` indica el directorio, archivo y extensión para guardar la imagen.
- `dpi`, del inglés *dots per inch*, determina la resolución de la imagen (siendo 300 buena calidad).
- `facecolor` y `edgecolor` son los colores a utilizar de fondo y en los bordes (blanco, `white 'w'`, por defecto).
- `format` es el `string` que indica el formato de la imagen ('pdf', 'png', 'svg', 'eps', etc.).
- `bbox_inches`, del inglés *bounding box*, indica el tamaño del marco a considerar alrededor del gráfico.

5.6. Representación con Pandas

Matplotlib es una librería muy versátil y potente para la producción de gráficos de muchos tipos, no solo para la representación de datos, sino también para la creación de diagramas y esquemas. Sin embargo, requiere conocimiento de una gran cantidad de funciones, con multitud de parámetros que pueden resultar complejos para el usuario medio.

Con el objetivo de disminuir la dificultad en la representación gráfica de datos, **Pandas** ofrece una interfaz para la creación de gráficos a partir de objetos `Series` y `DataFrames`. Para ello, ambos objetos tienen un método `plot()` que utiliza **Matplotlib** para visualizar los datos.

Por defecto, en una `Series` o `DataFrame`:

- El índice se utiliza como coordenada en el eje X.
- La lista de valores en una serie, o cada columna en un `frame`, describen una línea del gráfico (cada valor se utiliza como coordenada en el eje Y).
- En `frames`, el nombre de la columna determina la referencia en la leyenda.

El Programa 18 muestra cómo se representa un `DataFrame` visualmente en **Pandas**.

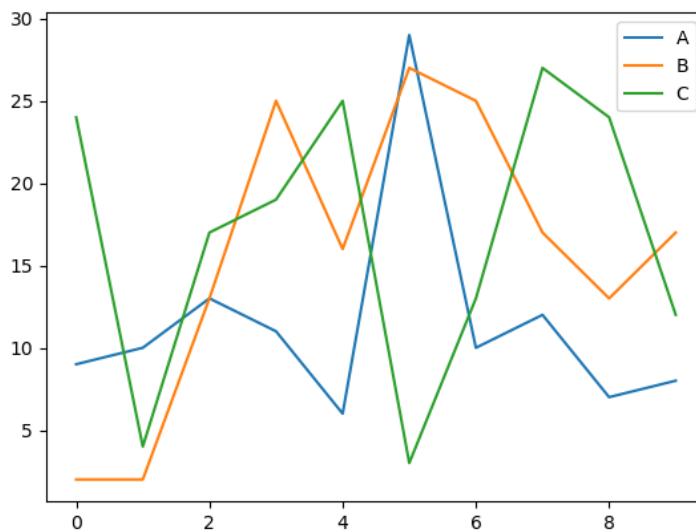
```
rand_matrix = np.random.randint(30, size=(10,3))

frame = pd.DataFrame(rand_matrix , columns=list('ABC'))

frame.plot()

plt.show() # aún es necesario llamar a show() para mostrar los gráficos
```

Programa 18. Uso de la interfaz gráfica de **Pandas**.



Salida 18. Resultado de representar un DataFrame con `plot()`.

Al igual que en **Matplotlib**, se pueden pasar argumentos a la función `plot()` para refinar el resultado de la figura.

Los parámetros más utilizados son:

- (`Series`) `label`: referencia a utilizar en la leyenda.
- `ax`: subfigura en **Matplotlib** en la que dibujar los datos (útil cuando se trabaja con múltiples subfiguras).
- `style`: `string` que define el estilo de la línea (más información en la tabla 6).
- `kind`: tipo de gráfico a representar ('`bar`', '`pie`', '`hist`', '`area`' '`line`', '`barh`', '`density`', '`kde`').
- `use_index`: `bool` que determina si utilizar el índice en los `ticks` del eje X.
- `xticks` e `yticks`: valores explícitos en los ejes X e Y.
- `title`: `string` a utilizar como título de la figura.
- (`DataFrame`) `subplots`: `bool` para indicar si se desean subfiguras separadas para cada columna.

La figura 6 muestra cinco tipos de gráficos obtenidos variando el parámetro *kind* en la representación de la siguiente serie en **Pandas**:

```
serie = pd.Series(np.arange(5), index=np.arange(-5, 0))
```

Los tipos utilizados son 'bar', 'pie', 'hist', 'area' y 'line'.

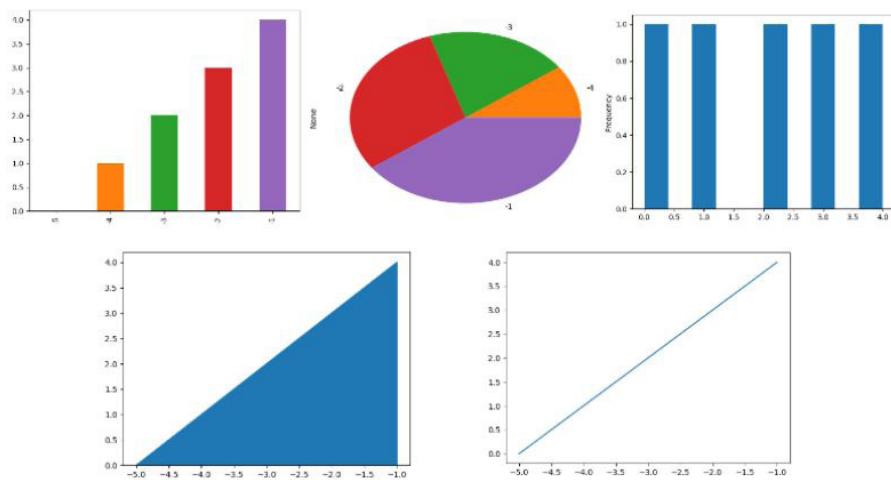
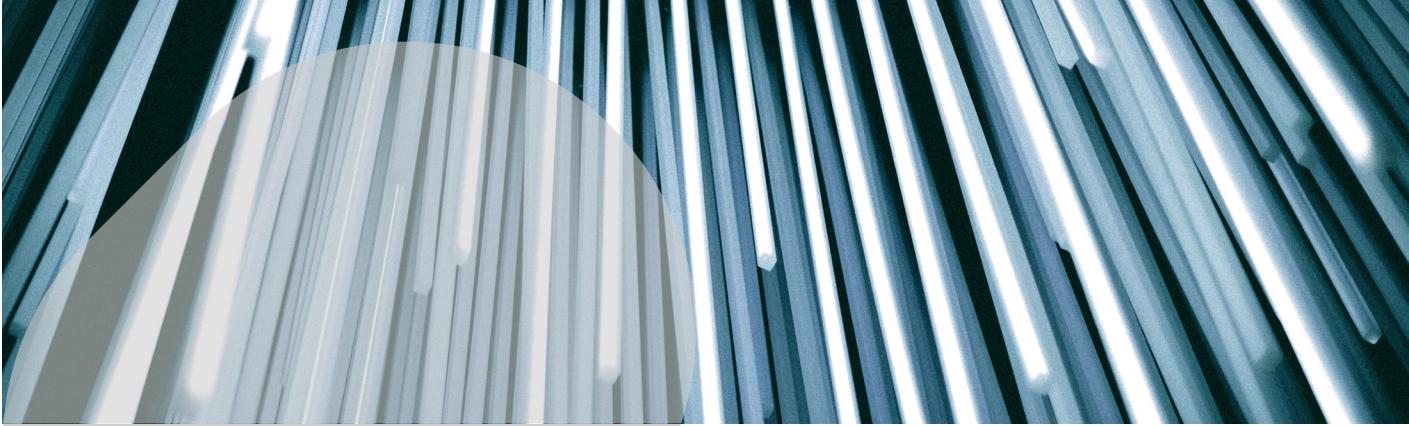


Figura 6. Diversas representaciones gráficas de una serie variando el parámetro *kind* del método `plot()`. De izquierda a derecha, y de arriba abajo: 'bar', 'pie', 'hist', 'area' y 'line'.



Capítulo 6

Python para ciencia de datos

En este capítulo vamos a describir tres aspectos fundamentales para el científico de datos en Python. En primer lugar, descubriremos cómo leer y escribir datos en el disco duro, importante para cargar los datos a nuestro modelo y guardar el progreso. También describiremos desde una perspectiva pragmática cómo manipular `strings`. Y finalmente introduciremos el tema de la [limpieza de datos](#) para su utilización tanto en el análisis de datos como en aprendizaje automático.

6.1. Lectura y escritura de archivos

Cargar y guardar datos es fundamental para evitar la transitoriedad del intérprete de Python. De poco serviría construir modelos predictivos con alta precisión si no fuésemos capaces de almacenarlos para su uso en producción. O sería poco productivo (o frecuentemente imposible) tener que generar los datos para entrenar nuestros modelos cada vez que quisieramos utilizar dicho modelo.

Hay varias formas de interactuar con el disco duro en Python, desde la escritura y la lectura directa con [streams](#), hasta el uso de las interfaces de **NumPy** y **Pandas**.

6.1.1. En Python

Como en cualquier lenguaje de programación, Python ofrece clases y métodos para trabajar con *streams* o flujos de datos, que pueden utilizarse para leer y escribir datos en el disco duro. La forma básica de abrir un *stream* es a través de la función `f = open(<path>)`, que retorna una referencia al *stream* que puede usarse para leer y escribir información sobre el archivo indicado en `<path>`. Una vez se ha terminado de trabajar con el archivo, se debe cerrar el *stream* con `f.close()`.

Para evitar tener que recordar cerrar el *stream*, se puede utilizar la construcción `with` en Python, que permite definir un alias temporal de la siguiente forma:

```
with open(<path>) as f:
```

```
    # trabajar con f
```

De esta forma, se puede trabajar con el *stream* definido como `f` sin necesidad de cerrarlo al final.

Para leer datos del *stream*, puede hacerse por líneas, iterando de forma natural:

```
for line in f:  
    # line contiene una línea en cada iteración
```



Dependiendo del formato del archivo y del sistema operativo, al leer una línea en un archivo de texto es posible que exista un *EOF character* (carácter indicador de final de línea). Python posee un método para eliminar dicho carácter de una `string`: `st.rstrip()`. Así, se pueden obtener todas las líneas de un *stream*, sin incluir EOF, filtrando de la siguiente forma:

```
lines = [x.rstrip() for x in f]
```

También es posible leer todas las líneas de una vez con `f.readlines()`, que retorna una secuencia con todas las líneas. Alternativamente, se puede solicitar la lectura de cierto número de caracteres con `f.read(5)`. En el caso de que sea adecuado hacerlo, se puede navegar a través del *stream* con las funciones `f.seek(<posición>)`, que mueve el puntero actual (posición actual de lectura) a la posición indicada, y `f.tell()`, que retorna la posición actual del *stream*.

Al abrir un *stream*, Python otorga un modo de acceso. Si no se define, por defecto el *stream* tiene solo modo de lectura, por lo que se puede leer información del archivo, pero no escribir. Para definir otro modo de acceso, se utiliza el segundo parámetro de la función `open(<path>, <mode>)`, donde `<mode>` es una de las `strings` indicadas en la Tabla 7.

| Modo de acceso | Descripción |
|----------------|--|
| r | Solo lectura |
| w | Solo escritura (borra el archivo si ya existe) |
| x | Solo escritura (falla si el archivo ya existe) |
| a | Crea un archivo (si existe lo abre y añade al final) |
| r+ | Lectura y escritura |
| b | Se puede añadir a otros modos para acceso binario |
| t | Modo texto para archivos de texto |

Tabla 7. Descripción de los modos de acceso en streams Python

Para escribir datos a un *stream* (abierto con el modo adecuado, 'w', 'x', 'a', o 'r+'), se pueden utilizar los métodos `f.write(<string>)` o `f.writelines(<secuencia>)` para escribir una `string` literal o una secuencia de `strings` (cada una en una línea) respectivamente.

Leer y escribir datos a un archivo de texto es suficiente en muchos casos. Sin embargo, dada la ubicuidad de los datos tabulares en big data y aprendizaje automático, es conveniente tener métodos para tratar con esta información directamente. Así, Python ofrece soporte para leer y procesar archivos CSV, estándar en la descripción de tablas.

Tanto para importar como para escribir datos tabulares en formato CSV, el módulo `csv` contiene funciones idóneas. `data = csv.read(<stream>, <delimiter>)` construye un lector de CSV iterable. Como los archivos CSV pueden tener los elementos separados por varios caracteres, el parámetro `<delimiter>` se utiliza para indicarlo. Cada iteración retorna una colección con los elementos de una línea de la tabla, en la que el elemento con índice 0 es la primera columna, etc. El Programa 19 contiene un ejemplo de cómo leer e iterar sobre los valores de una tabla CSV.

```
# Tabla1.csv contiene la Tabla 1 del manual, con valores separados por comas
import csv

f = open('Tabla1.csv')

data_reader = csv.reader(f, delimiter=',')
with open('Nueva_tabla.csv', 'w') as f2:
    csv_writer = csv.writer(f2, delimiter=';')
    for line in data_reader:
        print(line[0])
        csv_writer.writerow(line)
f.close()
```

Programa 19. Lectura, procesado y guardado de elementos en archivos CSV.

Operación

```
a + b
a - b
a / b
a // b
a * b
a ** b
a % b
a & b
a | b
a ^ b
a == b
a != b
a is b
a is not b
a <= b, a < b
a >= b, a > b
```

Salida 19. Resultado de la lectura del archivo CSV, en el que solo se imprimen los elementos de la primera columna.
Como resultado, el archivo CSV es copiado a 'Nueva tabla.csv' pero utilizando el carácter ; como separador.

6.1.2. En NumPy

NumPy ofrece su propia interfaz para leer y escribir ndarrays en disco. Por convención, estos archivos tienen el formato '.npy'. Para guardar cualquier ndarray array1 en el disco, se utiliza el método np.save(<filename>, array1).

También se pueden guardar múltiples ndarrays al mismo tiempo con np.savez(<filename>, a=arr1, b=arr2) sin comprimir, o np.savez_compressed(<filename>, a=arr1, b=arr2).

En cualquier caso, el método loaded = np.load(<filename>) carga la ndarray del archivo <filename>. Si el archivo contiene varias ndarrays, loaded es un diccionario con todas.

6.1.3. En Pandas

De la misma forma que **NumPy** puede grabar y leer ndarrays directamente del disco, **Pandas** ofrece formas de manipular datos tabulares. El método más común es pd.read_table, que genera un DataFrame o Series a partir de la información en el archivo CSV:

```
object = pd.read_table(<filename>, names=<nombres_de_columnas>, index_col=<-columna_como_index>, sep=<caracter_separador>, na_values=<string_na>)
```

Descripción de los parámetros:

- `names=<nombres_de_columnas>`: lista de strings a utilizar como nombres de columnas.
- `index_col=<columna_como_index>`: columna a utilizar como label o índice de las filas en el DataFrame o Series.
- `sep=<caracter_separador>`: carácter a utilizar como separador en el archivo CSV.
- `na_values=<string_na>`: lista de strings en el CSV que serán interpretados como valores nulos NaN. Puede también indicarse un diccionario en el que cada clave es el nombre de una columna, y el valor es una lista de elementos que se interpretarán como NaN en esa columna.

Para grabar en un archivo, tanto DataFrame como Series tienen un método:

```
to_csv(<filename>,           na_rep=<string_NaN>,           columns=<columnas_a_grabar>,
       sep=<separador>)
```

El parámetro `na_rep` se utiliza para indicar la string que **Pandas** utilizará en el CSV para valores NaN y nulos. `columns` sirve para seleccionar un subset de columnas a grabar (si no se desea grabar todo el objeto). `sep` es para indicar el carácter separador utilizado en el CSV.



Enlace de interés

Pandas también ofrece interfaces para leer archivos en formatos estándares como JSON, Excel o HDF5. Para más información, el enlace a continuación contiene la API de **Pandas** relacionada con input y output, incluyendo lectura y escritura de varios formatos:

<https://pandas.pydata.org/pandas-docs/stable/reference/index.html>

6.2. Trabajo con strings

En el tratado y procesamiento de datos, dada la naturaleza de muchos datos, es frecuente tener que manipular strings. Algunas de las tareas habituales son: limpiar strings de espacios blancos innecesarios, dividir strings en unidades, combinar strings o forzar un formato particular.

En Python, para utilizar caracteres de escape en strings, como tabulación (`\t`) o fin de línea (`\n`), se marcan con el carácter `\` primero. Esto quiere decir que, internamente, Python interpreta cualquier `\` en una string como un carácter de escape. No obstante, esto no es siempre cierto y el caso más común es cuando se desea expresar un directorio completo en Windows. Si se desea incluir el carácter `\` como tal en una string, se debe indicar como doble `\\"`. Otra alternativa es marcar la string con el prefijo `r`, lo que indica a Python que no hay caracteres de escape. Las dos strings siguientes son equivalentes:

```
s1 = 'c:\\windows\\user\\myfiles'
s2 = r'c:\\windows\\user\\myfiles'
```

Formar una `string` con una plantilla determinada es conveniente, por ejemplo, cuando se desea imprimir cierta información de forma homogénea. Para ello, Python incluye la función `<string>.format()`, que asocia marcadores dentro de la `string` con valores absolutos en los parámetros.



Ejemplo

```
# Uso de una plantilla para formar strings

template = "{0:.2f} {1:s} equivalen a {2:d} euros"

euros= template.format(9,'libras esterlinas',10)

dolares = template.format(11,'dolares',10)

# euros tiene el valor "9 libras esterlinas equivalen a 10 euros"

# dolares tiene el valor "11 dolares equivalen a 10 euros"
```

Manipular `strings` es sencillo:

- Para dividir una `string` en un listado de elementos mediante un separador, se puede utilizar el método `<string>.split(<separador>)`, que retorna dicho listado.
- Para unir elementos de una lista en una `string` con un carácter de por medio (operación inversa a 1), `<string_intermedia>.join(<listado>)` forma una `string` concatenando los elementos del `<listado>` y añadiendo `<string_intermedia>` entre medio.
- `<original>.replace(<string_a>,<string_b>)` reemplaza las ocurrencias de `<string_a>` en la `string` `<original>` con `<string_b>`.
- `<string>.strip()` elimina espacios en blanco a ambos lados de `<string>`.
- `<string>.upper()` y `<string>.lower()` convierten la `<string>` en mayúsculas y minúsculas respectivamente.

Otra tarea frecuentemente realizada es la de buscar un fragmento dentro de una `string`. Python permite comprobar la presencia con un simple `<substring> in <string>`, que retorna `True` cuando `<substring>` forma parte de `<string>`. Por ejemplo:

```
genre = 'comedy|drama|romantic'

'comedy' in gente # retorna True

'thriller' in genre # retorna False
```

También se puede averiguar el índice de la primera ocurrencia de un carácter con `<string>.index('a')`, que retorna un número con la posición de la primera ocurrencia, por ejemplo, para separar el dominio del nombre de usuario en una dirección de correo. `<string>.find(<carácter>)` funciona de manera similar pero retorna `-1` si no encuentra el carácter. Se puede averiguar el número de ocurrencias de un carácter con `<string>.count(<carácter>)`.

6.3. Limpieza de datos

Los modelos de predicción son tan buenos como la información que se usa para construirlos y entrenarlos (“garbage in, garbage out”). Por lo tanto, no es de extrañar que gran parte del tiempo que emplean los analistas de datos lo dediquen a explorar y transformar la información que poseen para llevarla a un estado en el que pueda ser utilizada.

Esta sección intenta ser una breve introducción a cómo realizar algunas de las tareas de limpieza de datos, y no una descripción exhaustiva de las técnicas o filosofías que hay detrás de ellas.

6.3.1. Aplicación de funciones a una colección

Cuando se poseen datos en cualquier colección de Python, en ocasiones se desea aplicar una medida global (por ejemplo, una función que escala los valores) a todos los elementos. Python posee tres tipos de funciones globales:

1. `map` aplica una función a todos los elementos de una colección y retorna una secuencia sin materializar con los resultados. La forma general es `list(map(<funcion>, <colección>))`.
2. `filter` retorna una secuencia sin materializar con solo los elementos de una colección que cumplen una condición establecida. La forma general es `list(filter(<funcion_booleana>, <colección>))`, donde `<funcion_booleana>` es una función que retorne `True` o `False`.
3. `Reduce` ejecuta una función con los elementos de una lista y retorna un escalar con el resultado. La forma general es `value = reduce(<funcion>, items)`.



La función `reduce` se encuentra en el módulo `functools`, por lo que requiere importación para ser utilizada.

`map`, `filter` y `reduce` son casos habituales en los que se suelen utilizar las funciones `lambda` para pasar funciones, ya que normalmente son expresiones cortas que no se reutilizan. El Programa 20 muestra ejemplos de las tres funciones globales.

```
from functools import reduce # necesario para reduce

# ejemplo de map para elevar todos los elementos al cuadrado
items = np.arange(10)

print(items)

squared = list(map(lambda x : x**2, items))

print(squared)

# ejemplo de filter para eliminar los elementos menores de 10
```

```
bigger_items = list(filter(lambda x : x > 10, squared))

print(bigger_items)

# ejemplo de reduce para obtener la suma de los elementos restantes

final_sum = reduce(lambda x,y : x + y, bigger_items)

print(final_sum)
```

Programa 20. Ejemplos de la aplicación de las tres funciones globales.

```
[0 1 2 3 4 5 6 7 8 9]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[16, 25, 36, 49, 64, 81]

271
```

Salida 20. map, filter y reduce usados como ejemplo de funciones globales sobre una colección.

6.3.2. Pandas para transformar datos

Cuando se trabaja en proyectos de big data, es raro encontrarse con datasets listos para ser utilizados directamente. Frecuentemente los datos requieren de procesos de transformación para tratar valores no definidos, columnas no estandarizadas, elementos duplicados, etc. **Pandas** incluye herramientas para facilitar el tratamiento y la limpieza de datos.

Valores no definidos

Recolectar datos es una tarea compleja y muchas veces inacabada, por lo que habitualmente se trabaja con datasets con valores incompletos. Para complicar las cosas, algunos algoritmos de aprendizaje automático y análisis de datos no toleran valores no definidos, por lo que estos huecos plantean un reto para el analista.

Hay varias formas de resolver valores no definidos

1. Eliminar datos con valores nulos. Pueden eliminarse elementos enteros si uno de sus valores es NaN. `<objeto>.dropna()` y filtra dichos elementos. En el caso de DataFrames, si uno de los valores en una columna es NaN, la fila entera queda eliminada. Si se desea ser menos restrictivo, por ejemplo, para solo eliminar las filas que no contengan al menos tres valores completos, puede indicarse con el argumento `thres=3`. Para eliminar solo los elementos con todos los valores NaN, se puede pasar el argumento `how='all'`.
2. Sustituir los valores NaN (fijo). En vez de eliminar elementos del dataset (particularmente difícil si la información es escasa), una alternativa es sustituir los valores NaN por valores fijos —el valor en concreto depende del problema en cuestión—. Tanto series como frames tienen el método `<objeto>.fillna(<valor>)`, con el cual se sustituyen todos los NaN por `<valor>`. El valor puede ser uno específico por columna, pasando como argumento un diccionario (en el que la clave es la columna y el valor es el elemento sustitutivo).

3. Sustituir los valores NaN (dinámico). Utilizando el argumento `method` de `<objeto>.fillna(method=<metodo>)`, es posible llenar los valores NaN de forma dinámica. Por ejemplo, '`backfill`' utiliza el valor inmediatamente siguiente al no definido para llenar el hueco, y '`ffill`' utiliza el valor anterior. Otra forma de llenar valores es mediante interpolación con `<objeto>.interpolate()`, que toma los valores anterior y siguiente, y los interpola para llenar el no definido.

Valores duplicados y reemplazos

Para encontrar valores duplicados, los objetos **Pandas** tienen el método `<objeto>.duplicated()`, que retorna un objeto de las mismas dimensiones con True, si el valor está duplicado, o False, si no lo está.

Para filtrar los elementos duplicados, se puede hacer a través del método `<objeto>.drop_duplicates()`, que es equivalente a `<objeto>[<objeto>.duplicated() == False]`.

También es conveniente tener la capacidad de reemplazar ciertos valores, como errores de medición. `Series` y `DataFrames` contienen el método `<objeto>.replace(<original>, <nuevo>)`, con el que pueden realizar dicho cambio. El valor `<nuevo>` puede ser un elemento único o una lista. Es posible pasar más de un recambio a la vez, pasando como argumento un diccionario, `<objeto>.replace(<diccionario>)`. En este caso, se reemplazan los elementos definidos como clave por los elementos valor.

Valores categóricos

Otra transformación fundamental es la conversión de valores continuos en categóricos. Para ello, **Pandas** ofrece un nuevo objeto: `Categorical`. Para construirlo, se definen las categorías (contenedores o `bins`) y los valores, y **Pandas** retorna un objeto `Categorical` indicando a qué categoría pertenece cada valor. La forma general es:

```
categorias = pd.cut(<valores>, <bins>)
```

Una vez construido, se puede hacer un recuento de todas las categorías con `pd.value_counts(<categoria>)`.

El argumento `<bins>` puede indicarse como un escalar y **Pandas** automáticamente divide los `<valores>` en segmentos de igual longitud. De forma similar, se puede utilizar la variante `pd.qcut(<valores>, <n-bins>)` para que Pandas divida los `<valores>` en tantas categorías como se determina en `<nbins>`, pero el rango se elige específicamente para repartir los valores de forma equitativa entre las categorías.



Ejemplo

Queremos categorizar la edad de los clientes en menor de edad (0-17), joven (18-34), adulto (35-64) y maduro (65-99).

```
bins = [0,18,35,65,99]  
  
edades = [2,16,25,18,33,71,44,54]  
  
categorias = pd.cut(edades,bins)
```

El objeto Categorical contiene la siguiente información, en la que cada elemento indica el segmento al que el valor en edades pertenece.

```
[(0, 18], (0, 18], (18, 35], (0, 18], (18, 35], (65, 99], (35, 65],  
(35, 65]
```

Categories (4, interval[int64]): [(0, 18] < (18, 35] < (35, 65] < (65, 99]]

En este caso, el recuento de categorías con pd.value_counts(categorias) da:

```
(18, 35]      3  
(0, 18]      3  
(35, 65]     2  
(65, 99]     1  
  
dtype: int64
```



Capítulo 7

Generadores de Datos

Cuando se trabaja con conjuntos de datos con un gran volumen de información se presenta un problema complicado de resolver. Estos conjuntos de datos necesitan de mucha cantidad de memoria que los contenga y esto se convierte en un cuello de botella. Para solucionar este problema se usan los generadores, que no son más que funciones que, en lugar de cargar los datos en un solo paso ocupando la memoria, los van generando a medida que se van necesitando. Así se ahorra memoria y permiten dividir el tiempo de procesamiento entre las sucesivas peticiones de datos.

7.1. Generadores en Python

Las funciones generadoras en Python son un tipo especial de función que devuelve un iterador *lazy*, objetos sobre los que se puede iterar como si fuera una lista o cualquier iterador común. Sin embargo, al contrario que las listas, los iteradores *lazy* no almacenan su contenido en memoria. Al programarlos, se hace como una función normal, pero se compilan de una forma especial en un objeto que soporta el protocolo de iteración, y cuando se llaman, no devuelven un resultado, sino que devuelven un generador que puede usarse para iterar sobre él.

Al contrario que las funciones normales que retornan un valor y salen, las funciones generadoras se suspenden y reanudan su ejecución, y guardan su estado en cada iteración. La clave está en el uso de `yield` en lugar del `return` habitual. Esta sentencia `yield` en una función suspende su ejecución y devuelve un valor de vuelta a quien la llama, pero guarda el estado de la función para que pueda continuar desde ese punto cuando se vuelve a llamar. Cuando esto sucede, la función continúa su ejecución inmediatamente tras el último `yield`. Este mecanismo permite a la función producir una serie de valores a lo largo del tiempo, en lugar de producirlos todos a la vez y devolverlos en alguna estructura como una lista.



Ejemplo

```
def squares(n):  
  
    for i in range(n):  
  
        yield i ** 2
```

Se definen como una función normal y genera valores cuadrados hasta un valor determinado. Produce un valor y lo devuelve a quien la llama, cada vez a través del bucle. Cuando se vuelve a ejecutar, se restaura su estado, incluidos los valores de sus variables y la ejecución vuelve inmediatamente tras el `yield`.

```
# Se usa el generador squares  
  
for i in squares(5):  
  
    print(f"{i}")
```

Programa 21. Uso de un generador dentro de un bucle.

0 1 4 9 16

Salida 21. Resultado del uso del generador.

Al usarla desde un bucle, la primera llamada arranca la función y consigue el primer valor; luego el control vuelve a la función justo tras la sentencia `yield` cada vez, en cada iteración del bucle.

Para acabar la generación de valores, las funciones usan una sentencia `return` sin valor, o simplemente dejan que el control de la ejecución acabe con la función.

Por debajo, el bucle externo que llama a la función `squares`, está haciendo uso del protocolo de iteración de Python, que determina que un generador tiene que implementar el método `__next__` que arranca una función o la reanuda desde donde produjo un valor la última llamada.



Ejemplo

```
>>> x = squares(3)

>>> next(x)

0

>>> next(x)

1

>>> next(x)

4

>>> next(x)

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Se puede usar el método `next` para hacer uso de un iterador y poder ver su ejecución paso a paso. En este caso, se puede ver que cada vez que se llama a `next` sobre el generador produce un valor y si se intenta producir uno más cuando ya ha acabado, produce una excepción.

Además del método `next` para obtener un resultado de un generador, se puede usar el método `send` que actúa de igual manera, pero es capaz de enviar información hacia el generador en cada invocación.



Ejemplo

```
def gen():
    for i in range(10):
        x = yield i
        print(f"{x} inside Gen")
```

La sentencia `yield` es capaz de recibir y devolver valores. Cuando se usa a través de `next` se envía `None` al generador, pero con `send` se pueden enviar valores en cada iteración.



Ejemplo

```
>>> g = gen()  
  
>>> next(g)  
  
0  
  
>>> g.send(20)  
  
20 inside Gen  
  
1  
  
>>> g.send(10)  
  
10 inside Gen  
  
2  
  
>>> next(g)  
  
None inside Gen  
  
3
```

7.1.1. Expresiones generadoras

Tal como las comprehension lists, las expresiones generadoras permiten crear rápidamente un objeto generador en unas pocas líneas de código. Se pueden crear sin construir y mantener en memoria el objeto entero antes de la iteración.



Ejemplo

```
squares_cl = [num**2 for num in range(5)]  
  
squares_cl  
  
Output:  
  
[0, 1, 4, 9, 16]
```

Esta comprehension list genera una lista con los cuadrados de los primeros cinco números. En cambio, creando una expresión generadora se crea un generador que puede usarse como un generador normal: simplemente usando los paréntesis en lugar de los corchetes al crear el objeto.



Ejemplo

```
squares_ge = (num**2 for num in range(5))

squares_ge

Output:

<generator object <genexpr> at 0x7f3acf2beeb0>
```

Se comporta exactamente igual que una función generadora.



Ejemplo

```
>>> next(squares_ge)

0

>>> next(g)

1
```



Las expresiones generadoras se usan como las `comprehension lists`, pero cambiando el corchete por el paréntesis.

7.1.2. Crear pipelines de datos con generadores

Los canales o *pipelines* de datos permiten encadenar código para procesar conjuntos de datos de gran tamaño o hacer *streaming* de datos sin agotar la memoria del sistema. Por ejemplo, se tiene un fichero CSV que se quiere procesar y se quiere alimentar el método que realiza dicho procesamiento sin tener que cargar todo el fichero a memoria de antemano, sino hacerlo poco a poco, leyendo y procesando una línea cada vez.

Para ello es interesante plantear una estrategia inicial.

- Se lee una línea del fichero.
- Se divide cada línea en una lista de valores.
- Se extraen los nombres de las columnas.
- Se usan los nombres de las columnas para crear un diccionario con los datos de cada línea.
- Se filtran los datos en base a algún criterio.
- Se calcula algún valor para dichos criterios.

Esto se puede realizar con **Pandas** o alguna otra librería de tratamiento de datos, pero también se puede gestionar usando generadores.

El primer paso es acceder al fichero y leer una línea del mismo usando una expresión de generador. El fichero CSV de ejemplo es una lista de los pasajeros embarcados en el Titanic, algunas de sus características y si sobrevivieron o no.



Enlace de interés

Para más información sobre el dataset referido, incluido un enlace de descarga, se puede acceder a la página de Kaggle:

<https://www.kaggle.com/c/titanic>



Ejemplo

```
file_name = "titanic.csv"

lines = (line for line in open(file_name))
```

Después, se crea otra expresión de generador a partir de la anterior, para dividir cada línea en una lista y poder acceder a los campos de cada línea.



Ejemplo

```
list_line = (s.rstrip().split(",") for s in lines)
```

Este nuevo generador `list_line` itera sobre el primer generador `lines`. Esto es un patrón común al crear *pipelines*: ir encadenando generadores. Ahora se piden los nombres de columna del CSV. Puesto que los nombres de columna suelen estar en la primera línea de los CSV, se pueden conseguir llamando a `next` de este segundo generador.



Ejemplo

```
cols = next(list_line)
```

```
cols
```

Output:

```
[ 'PassengerId',
```

```
    'Survived',
```

```
    'Pclass',
```

```
    'Name',
```

```
    'Sex',
```

```
    'Age',
```

```
    'SibSp',
```

```
    'Parch',
```

```
    'Ticket',
```

```
    'Fare',
```

```
    'Cabin',
```

```
    'Embarked' ]
```

Esta primera llamada a `next` avanza el generador sobre `list_line` una vez, devuelve una lista con los valores de la primera línea y guarda el estado para siguientes iteraciones.

Para ayudar a filtrar y operar sobre los datos, se crean diccionarios donde las claves son los nombres de las columnas del CSV, usando otra expresión de generador.



Ejemplo

```
passengers = (dict(zip(cols, data)) for data in list_line)
```

Este generador itera a través de las listas que produce `list_line`, y usa `zip` y `dict` para crear un diccionario con las columnas como claves y los datos de cada línea como valores.

A continuación, se crea un nuevo generador para filtrar los datos como se desea.



Ejemplo

```
passenger_in_class3_survived = (int(passenger["Survived"]) for passenger in passengers if int(passenger["Pclass"]) == 3)
```

Esta nueva expresión de generador itera sobre los resultados del generador anterior, `passengers`, y devuelve el valor del campo `Survived` (que es 0 si el pasajero murió o 1 si sobrevivió), pero solo cuando el valor del campo `Pclass` sea 3, es decir, que el pasajero compró un billete de clase 3.

Hasta ahora no se ha iterado realmente sobre todos esos datos, hasta que no se use una expresión de generador en un bucle o en una función que funcione sobre objetos iterables. Para obtener todos los pasajeros de tercera clase que sobrevivieron al naufragio, se usa la función `sum` que trabaja sobre un iterable; en este caso sobre el último generador.



Ejemplo

```
passengers_in_class3_survived = sum(passenger_in_class3_survived)
passenger_in_class3_survived
Output:
119
```

Es en este punto, cuando la función `sum` desencadena toda la cascada de generadores y pone en marcha el pipeline de datos, iterando sobre el generador `passenger_in_class3_survived`, el cual itera sobre el anterior, y así sucesivamente, hasta que se produce la suma total de los datos filtrados.

7.2. Generadores de Python para entrenar modelos

Estos generadores se pueden usar fácilmente para alimentar el entrenamiento de redes neuronales. Por norma general, estas redes neuronales necesitan de conjuntos de datos gigantes para poder trabajar y estos habitualmente consumen demasiada memoria para poder ser trabajados de una sola vez. La solución es alimentar a dichas redes neuronales a través de generadores.

Se tiene un supuesto fichero de 500 000 000 líneas con datos: probablemente no se podría cargar en memoria para poder procesarlo. Por ejemplo, este fichero podría tener un mapeado lineal, en el cual a cada valor de x le corresponde un valor igual a y , es decir, la función $y: f(x) = x$. Una de los escenarios de regresión más simples que se pueden encontrar.



Ejemplo

x, y

1, 1

2, 2

3, 3

Es sencillo crear un fichero con estas características en Python y ocuparía varios GB de memoria. Por tanto, fuera del alcance para cargarlo de una sola vez y usarlo como conjunto de datos de entrenamiento para una red neuronal. Pero se puede definir un tamaño de *batch* o lote que sea de un tamaño manejable y alimentar a la red con pequeños trozos de dicho fichero, usando un generador.

```
def generate_arrays_from_file(path, batch_size):  
    inputs = []  
    targets = []  
    batchcount = 0  
  
    while True:  
        with open(path) as f:  
            for line in f:  
                x,y = line.split(',')  
                inputs.append(x)  
                targets.append(y)  
                batchcount += 1  
  
                if batchcount > batchsize:
```

```
X = np.array(inputs, dtype='float32')  
  
y = np.array(targets, dtype='float32')  
  
yield (X, y)  
  
inputs = []  
  
targets = []  
  
batchcount = 0
```

Programa 22. Función generadora para alimentar un entrenador.

Se puede crear una función generadora que reciba como parámetros la ruta donde se encuentra el fichero a procesar y el tamaño de cada *batch* que se quiere entregar en cada iteración del generador. La función crea un par de listas vacías donde guardar las entradas y los targets para el entrenamiento. Va leyendo del fichero líneas y separa los valores y añade el primero a la lista de entradas y el segundo a la de targets. Cuando ha llegado a las líneas para el lote actual, entonces convierte las dos listas de entradas y *targets* en dos arrays de **numPy**, y los devuelve en una tupla, mediante un *yield*. En la próxima ejecución, esta función primero limpia las listas temporales, reinicia el contador del lote y sigue leyendo el fichero en el punto en el que se quedó anteriormente. De esta manera, va generando arrays de 250 elementos en lotes, poco a poco.

A la hora de usar este generador para entrenar una red neuronal, se le puede pasar a la función *fit* dicho generador como fuente de datos.



Ejemplo

```
num_rows = 5e8  
  
batch_size = 250  
  
model.fit(generate_arrays_from_file('./five_hundred.csv', batch_size),  
steps_per_epoch=num_rows / batch_size, epochs=10)
```

En este ejemplo, se usa un modelo `Sequential` de la librería **Keras**. Dicho modelo usará el generador y le irá pidiendo lotes para entrenar la red. Estos generadores deben proveer una tupla compuesta por dos elementos: el primero, los *inputs* y el segundo, los *targets*; y si es necesario un tercer elemento, con los *sample_weights*. Se pueden usar este tipo de generadores también para generar el conjunto de datos de validación.

7.3. Generadores de datos de imágenes con Keras

La librería **Keras** tiene una clase llamada `ImageDataGenerator` que en su interior es un generador que permite acceder a lotes de imágenes directamente desde el sistema de archivos, por lo que se pueden entrenar redes neuronales con conjuntos de datos de gran tamaño, siempre que se disponga de espacio en disco suficiente.

El requisito principal es que las imágenes se encuentren organizadas en una estructura de directorios específica, donde todas las imágenes correspondientes a una misma categoría estén ubicadas en una carpeta con el nombre de dicha clase.

Ejemplo

```
└── airplane
    ├── air-air-travel-airbus-aircraft-358319.jpg
    ├── flight-flying-airplane-jet-40753.jpg
    ...
└── boat
    ├── high-angle-photo-of-white-boat-on-body-of-water-1295036.jpg
    ├── person-standing-on-dock-beside-boat-531474.jpg
    ...
└── car
    ├── action-asphalt-auto-automobile-210019.jpg
    ├── blue-sedan-712618.jpg
    ...
└── truck
    ├── 4k-wallpaper-4x4-auto-automobile-1149058.jpg
    ├── auto-automobile-automotive-blur-528426.jpg
    ...
```

En este ejemplo hay cuatro categorías, *airplane*, *boat*, *car* y *truck*. Y dentro de cada carpeta hay una serie de imágenes que corresponden a cada categoría.

El generador `ImageDataGenerator` tiene varias maneras de proveer las imágenes: en este caso, se va a usar desde un directorio y para ello se usa el método `flow_from_directory`.

Ejemplo

```
data_generator = ImageDataGenerator()

generator = data_generator.flow_from_directory('resources', batch_
size=1, target_size=(299, 299))
```

Dicho método `flow_from_directory` como primer parámetro toma la ruta donde están las imágenes estructuradas en carpetas con sus respectivas clases, el tamaño de los *batches* o lotes y el tamaño al que se van a redimensionar las imágenes una vez leídas para alimentar el entrenamiento de manera homogénea.

Se puede preguntar al generador acerca de las clases (subcarpetas) que ha reconocido a partir de las carpetas con la propiedad `class_indices`.



Ejemplo

```
generator.class_indices  
  
Output:  
  
{'airplane': 0, 'boat': 1, 'car': 2, 'truck': 3}
```

Devuelve un diccionario con cada una de las clases que ha reconocido junto con un índice para cada una de ellas. Se puede usar como un generador normal y con `next` se puede acceder a las imágenes.

```
n_images = 24  
  
while n_images > 0:  
  
    images, labels = next(generator)  
  
    plt.imshow((images[0].astype(int)))  
  
    plt.show()  
  
    print(f'Etiqueta {labels[0]}')  
  
    n_images -= 1
```

Programa 23. Pintado de las imágenes leídas por el generador.

```
Etiqueta [0. 0. 1. 0.]
```

```
Etiqueta [0. 1. 0. 0.]
```

```
...
```

Salida 23. Resultado de los índices de un par de imágenes.

El generador en este caso devolverá una tupla con las imágenes del lote y las etiquetas del mismo. En este ejemplo, como se ha usado un tamaño de *batch* de un solo elemento, el generador irá devolviendo lotes de un solo elemento. Se puede acceder a dicha imagen y mostrarla por pantalla con **Matplotlib** e imprimir la etiqueta, que determina a qué categoría pertenece cada imagen.

Estos generadores se pueden usar de igual manera para entrenar redes en el método `fit` del modelo como primer parámetro si es el conjunto de datos para entrenamiento, y en el parámetro `validation_data` si se usa para la validación del modelo.



Ejemplo

```
model.fit(generator, epochs = epochs, validation_data = generator_val)
```

7.3.1. Flow

Además de alimentar al generador a través de imágenes en un directorio, usando el método *flow* se puede hacer con arrays de **numPy**.

Como ejemplo se usa el conjunto de datos MNIST, que contiene una base de datos de imágenes de dígitos escritos a mano y que viene como paquete dentro de **Keras**.



Ejemplo

```
from keras.datasets import mnist  
  
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Se llama a *load_data* del paquete *mnist*, lo cual nos devuelve los conjuntos de datos ya divididos para entrenamiento, en los datos y los *targets*, y en estructuras de datos *ndarray* de **numPy**.



Ejemplo

```
data_generator = ImageDataGenerator()  
  
training_generator = datagen.flow(x_train, y_train, batch_size=64)
```

Se crea un generador a partir de la clase *ImageDataGenerator*, y posteriormente se usa el método *flow* para crear el generador a partir de los arrays de **numPy** cargados previamente, para los datos y para los *targets*.

Si los datos estuvieran en un *dataframe* de la librería **Pandas** se tendría que hacer uso del método *flow_from_dataframe*, que requiere como primer parámetro un *dataframe* donde deberían estar las rutas a las imágenes que se van a usar para alimentar el entrenamiento.

7.3.2. Augmentations

Para entrenar a las redes neuronales es importante tener conjuntos de datos muy grandes y, además, es interesante exponer a los modelos a aspectos ligeramente distintos de los datos, para poder generalizar mejor. Para hacer esto se pueden recolectar esos datos manualmente o generarlos a partir de datos ya existentes aplicando algunas transformaciones. Este método se denomina *data augmentation*, o incremento de los datos.

Usando la clase `ImageDataGenerator` de **Keras** se pueden realizar dichas transformaciones a la hora de alimentar los modelos a partir de un conjunto de datos de imágenes. Esta clase tiene una serie de parámetros para realizar dichas transformaciones que le pueden proveer al crear dicha clase. Se pueden realizar varias de estas transformaciones juntas para dar más riqueza a los datos generados.



Enlace de interés

Imagen de una suricata de prueba.

<https://www.pexels.com/photo/brown-animal-1454786/>

- `rotation_range`. Gira cada imagen al ángulo especificado.

```
data_generator = ImageDataGenerator(rotation_range = 45)
```



Figura 7. Augmentation de 6 imágenes con rotación.

- `width_shift_range`, `height_shift_range`. Generan un desplazamiento horizontal/vertical. Si el valor es un *float* menor que 1, se considera un porcentaje. En caso de un valor mayor que 1, se considera que este valor es el número de píxeles que se quiere mover.

```
data_generator = ImageDataGenerator(width_shift_range = 0.2)
```

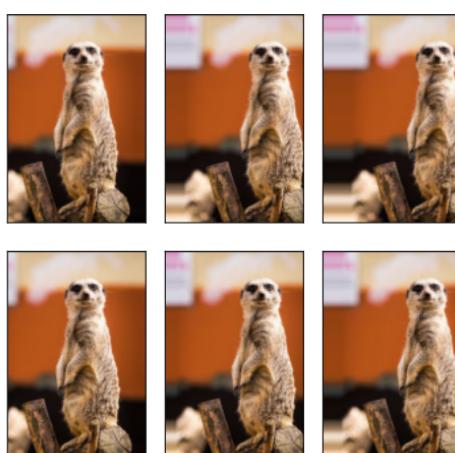


Figura 8. Augmentation de 6 imágenes con desplazamiento.

- `brightness_range`. Modifica el brillo de la imagen; se le pasa el rango mínimo y máximo del brillo.

```
data_generator = ImageDataGenerator(brightness_range = [0.1, 0.9])
```

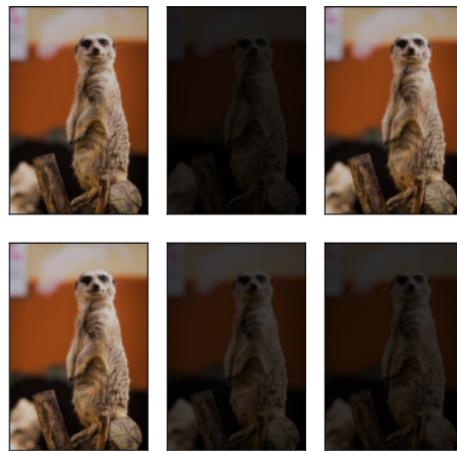


Figura 9. Augmentation de 6 imágenes con brillo.

- `Rescale`. Normaliza los valores de los píxeles a un rango específico. Para imágenes de 8 bits se suele reescalar a 1/255 para tener estos valores de color entre 0 y 1.

```
data_generator = ImageDataGenerator(rescale = 1/255.)
```

- `shear_range`. Se le pasa el ángulo en grados que se le quiere aplicar al `shear`.

```
data_generator = ImageDataGenerator(shear_range = 45)
```

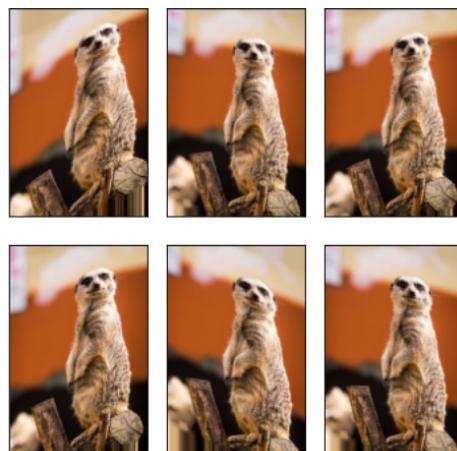


Figura 10. Augmentation de 6 imágenes con shear.

- `zoom_range`. Hace un zoom de la imagen.

```
data_generator = ImageDataGenerator(zoom_range = 0.5)
```

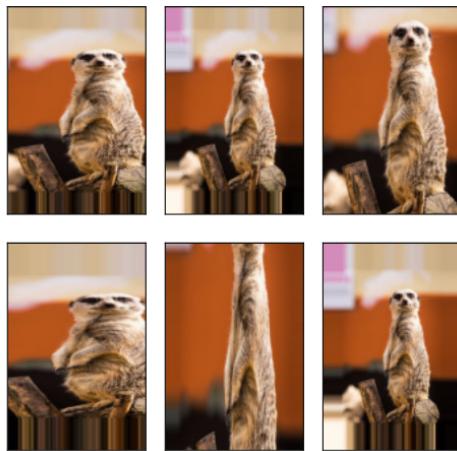


Figura 11. Augmentación de 6 imágenes con zoom.

- `channel_shift_range`. Cambia los valores de los canales de manera aleatoria por los valores suministrados.

```
data_generator = ImageDataGenerator(channel_shift_range = 100)
```

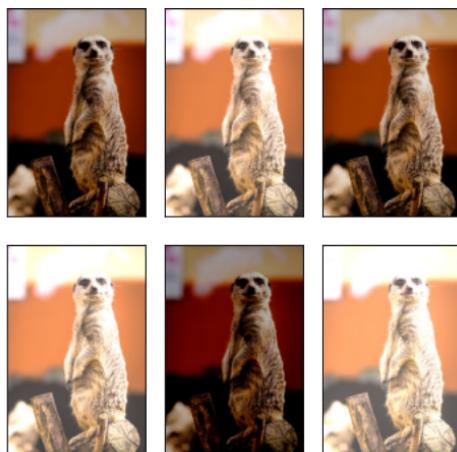


Figura 12. Augmentación de 6 imágenes con canales.

- `horizontal_flip`, `vertical_flip`. Dan la vuelta a la imagen en horizontal o vertical.

```
data_generator = ImageDataGenerator(vertical_flip = True)
```

- `preprocessing_function`. Se le aplica una función a cada imagen antes del paso del `augmentation`.



Ejemplo

```
def blur(img):  
  
    return (cv2.blur(img, (5,5)))  
  
data_generator = ImageDataGenerator(preprocessing_function = blur)
```

7.4. Generadores de datos custom con Keras

Es factible y muchas veces deseable ir un paso más allá y crear generadores de datos *custom* a medida, para poder tener un control más avanzado del proceso de generación de estos datos. La clase de **Keras Sequence** es la raíz de la cual deben heredar los generadores de datos y tiene algunos métodos que se deben sobreescribir para implementar un cargador de datos a medida.

```
from keras.models import Sequential  
  
import tensorflow as tf  
  
class CustomDataGenerator(tf.keras.utils.Sequence):  
  
    def __init__(self, batch_size, *args, **kwargs):  
  
        self.batch_size = batch_size  
  
        pass  
  
    def __len__(self):  
  
        # Returns number of batches per epoch  
  
        return total_data / self.batch_size  
  
    def __getitem__(self, index):  
  
        # Generate one batch of data  
  
        return X, y  
  
    def on_epoch_end(self):  
  
        pass
```

Programa 24. Prototipo de generador de datos custom.

Usar un generador a medida es bastante sencillo. Solo hay que pasar dicho generador al método `fit` del modelo, tanto el de entrenamiento como el de validación, si es necesario.



Ejemplo

```
train_ds = CustomDataset(...)

valid_ds = CustomDataset(...)

model.fit_generator(train_ds, validation_data=valid_ds)
```

Se va a construir un generador a medida como ejemplo para poder comprender cómo debe funcionar en líneas generales. En el constructor del generador se suelen inicializar los valores de configuración.



Ejemplo

```
def __init__(self, data, batch_size, dim, n_classes, shuffle):
    self.dim = dim
    self.data = data
    self.batch_size = batch_size
    self.list_IDs = np.arange(len(data))
    self.n_classes = n_classes
    self.shuffle = shuffle
    self.on_epoch_end()
```

En este caso el argumento `data` se refiere a un `dataframe` que contiene la secuencia de `paths` de `frames` y sus correspondientes etiquetas y atributos. `self.list_IDs` es una lista de todos los índices de los datos que se usarán para cargar los datos en lotes.

Al método `on_epoch` se le llama una vez al inicio y luego tras cada epoch del ciclo de entrenamiento, y usualmente se usa para mezclar los índices de los datos.



Ejemplo

```
def on_epoch_end(self):  
  
    self.indexes = self.list_IDs  
  
    if self.shuffle == True:  
  
        np.random.shuffle(self.indexes)
```

En el ejemplo se mezcla el orden de las filas del *dataframe*. Otro método que hay que implementar es `__len__`, que debe devolver el número de pasos en un *epoch*, por ejemplo, usando el número de muestras y el tamaño del *batch*.



Ejemplo

```
def __len__(self):  
  
    return int(np.floor(len(self.data) / self.batch_size))
```

Otro método imprescindible es `__getitem__`. Recibe como parámetro una posición, el índice, y la combina con el tamaño de los *batches* para devolver un lote de datos a partir de esa posición.



Ejemplo

```
def __getitem__(self, index):  
  
    index = self.indexes[index * self.batch_size : (index+1) * self.  
    batch_size]  
  
    list_IDs_temp = [self.list_IDs[k] for k in index]  
  
    X, y = self.__get_data(list_IDs_temp)  
  
    return X, y
```

Se obtienen los índices del *batch* que se pide, y la lista de elementos que van a formar el *batch*, y luego se delega a otra función la generación del conjunto de datos para este *batch*. Se podría hacer directamente en la misma función, pero se considera buena práctica hacerlo en otro bloque de código.

La función `__get_data` se va a encargar de generar este subconjunto de datos. Se define como privado: puesto que no forma parte de los miembros de la clase `Sequence` de la cual se hereda, puede llamarse de cualquier manera.

```
def __get_data(self, list_IDs_temp):  
    X_data = []  
    y_data = []  
  
    for i,_ in enumerate(list_IDs_temp):  
        seq_frames = self.data.iloc[i, 0]  
        y = self.data.iloc[i, 1]  
  
        temp_data_list = []  
  
        for img in seq_frames:  
            try:  
                image = cv2.imread(img, 0)  
                ext_img = cv2.resize(image, self.dim)  
            except Exception as e:  
                #Code for handling exceptions  
                temp_data_list.append(ext_img)  
  
    X_data.append(temp_data_list)  
    y_data.append(y)  
  
    X = np.array(X_data)  
    y = np.array(y_data)  
  
    return X, keras.utils.to_categorical(y, num_classes=self.n_classes)
```

Programa 25. Función donde generar un conjunto de datos para un lote.

Adicionalmente, se pueden añadir rutinas de preprocessamiento o *augmentation* en tiempo real. El método recibe como argumento la lista temporal de índices y la usa para indexar el *dataframe* y acceder a la correspondiente lista de *frames* y sus etiquetas. Esos *frames* se leen usando **OpenCV** y se guardan en una lista, que luego se convierte a un array de **numPy** antes de devolverlas.

En esta función se puede poner cualquier origen de datos que se necesite, desde cargar imágenes, cargar texto, o ambas cosas simultáneamente, o cualquier otro tipo de datos.

Para usarlo se definen los parámetros para el generador y se usan para crear instancias para el entrenamiento y la validación, que luego se pasan al método *fit* para realizar el entrenamiento usando el generador de datos a medida creado.



Ejemplo

```
params      =      { 'batch_size':64,      'dim':(48,48),      'n_classes':2,
'shuffle':True }

train_generator = DataGenerator(path_to_traindata,**params)

validation_generator = DataGenerator(path_to_validationdata,**params)

model.fit(train_generator,  epochs,  validation_data=validation_generator)
```

Capítulo 8

Análisis de datos textuales con NLTK

En la mayoría de las ocasiones se trabaja con datos numéricos o en forma tabular, fácilmente procesables a través de expresiones matemáticas o técnicas estadísticas. Pero en otras, los datos se componen de texto, y responden a reglas gramaticales que varían entre distintas lenguas. En estos textos, las palabras, sus significados y las emociones que transmiten pueden ser una fuente útil de información.

Últimamente, con la llegada del big data y la inmensa cantidad de datos textuales que vienen de internet, se han desarrollado multitud de técnicas de análisis de texto. De hecho, estos datos suelen ser muy difíciles de analizar, aunque al mismo tiempo son una fuente significativa de información útil, aumentado además por la gran cantidad disponible de dichos datos (por ejemplo, la literatura que se genera, los posts que se publican en internet, comentarios en redes sociales y chats, etc.).

Por tanto, analizar dichos textos es una fuente de interés, algunas técnicas habituales son:

- Análisis de la distribución de la frecuencia de palabras
- Reconocimiento de patrones
- Etiquetado
- Análisis de enlaces y asociaciones
- Análisis de sentimientos

8.1. Natural Language Toolkit NLTK

La librería **NLTK** incluye multitud de herramientas especializadas en el procesamiento y análisis de datos textuales. Esta librería también incluye multitud de textos de ejemplo, llamados *corpora*. Esta colección de textos surge de la literatura, y son una buena base para practicar las técnicas desarrolladas con dicha librería, se suelen usar para realizar tests.

Instalar **NLTK** es sencillo usando el gestor de paquetes *standard* de Python pip.

```
pip install nltk
```

8.2. Importar la librería NLTK y la herramienta de descargas

Para usar la librería primero hay que importarla.

```
import nltk
```

Si se quiere importar texto de la colección *corpora* existe una función llamada `nltk.download_shell()`, que abre la herramienta NLTK Downloader y permite realizar selecciones a través de una serie de opciones guiadas.

```
nltk.download_shell()
```

Si se usa una sesión interactiva o un *notebook*, en este momento, la herramienta espera que el usuario introduzca una opción de las listadas. Haciendo uso de la opción `I` se obtiene una lista de los posibles paquetes de **NLTK** que se pueden descargar para extender su funcionalidad, incluyendo los textos de la colección *corpora*.

Ejemplo

```
d) Download l) List u) Update c) Config h) Help q) Quit
```

```
Packages:
```

```
[ ] abc..... Australian Broadcasting Commission 2006  
[ ] alpino..... Alpino Dutch Treebank  
[ ] averaged_perceptron_tagger Averaged Perceptron Tagger  
...
```

```
Hit Enter to continue:
```

Para trabajar con algunos ejemplos y aprender a usar la librería, se recomienda descargar el corpus Gutenberg, que es una pequeña selección de textos extraídos del archivo electrónico con más de 25 000 libros Proyecto Gutenberg (<http://www.gutenberg.org/>).

Para descargar dicho paquete hay que usar la opción d, y contestar `gutenberg` cuando la herramienta pregunte por el nombre del paquete a descargar.

Ejemplo

```
Download which package (l=list; x=cancel) ?  
  
>>> gutenberg  
  
Downloading package gutenberg to /home/nltk_data...  
  
Unzipping corpora/gutenberg.zip.
```

También se puede usar el comando `download` pasando como argumento el nombre del paquete a descargar para hacerlo directamente.

```
nltk.download('gutenberg')
```

Para acceder al paquete descargado se accede a través de la propiedad `corpus` de `nltk`, y se pueden ver los contenidos usando la función `fileids`, la cual muestra los nombres de los ficheros que contiene.

Ejemplo

```
gb = nltk.corpus.gutenberg  
  
print(gb.fileids())  
  
Output:  
  
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt', 'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt', 'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

Para acceder al contenido de alguno de dichos ficheros hay que seleccionar uno de ellos, por ejemplo, ***Alice in Wonderland*** de **Lewis Carroll**, asignándolo a una variable. Un modo de extracción es para palabras, es decir, se crea una lista de palabras como elementos usando la función `words`.

```
alice = nltk.corpus.gutenberg.words('carroll-alice.txt')
```

Obtener la longitud del texto en palabras de dicha obra se puede hacer usando la función `len`. En este caso, la obra tiene 34 110 palabras.

```
len(alice)
```

La variable `alice` es una lista que contiene todas las palabras del texto, a la cual se puede acceder de la manera habitual en Python.

Ejemplo

```
alice [:12]
```

Output:

```
[['[', 'Alice', "", 's', 'Adventures', 'in', 'Wonderland', 'by',
'Lewis', Carroll', '1865',']]
```

En ocasiones puede resultar interesante cargar el fichero con una estructura más jerarquizada, es decir, tener las frases del texto a un primer nivel y luego cada frase dividida por palabras. Para ello hay que cargar el paquete `punkt`, que se encarga de tokenizar los textos.

```
nltk.download('punkt')
```

Se puede cargar el fichero de texto a una lista usando frases en lugar de palabras. Para ello se usa la función `sents`, para obtener una lista más estructurada, con frases en el primer nivel de la lista, y cada frase a su vez es una lista de las palabras que contiene.

Ejemplo

```
alice_s = nltk.corpus.gutenberg.sents('carroll-alice.txt')
```

```
alice_s[2]
```

Output:

```
['Down', 'the', 'Rabbit', '–', 'Hole']
```

8.3. Concordancia y colocaciones

Una de las cosas más básicas que se pueden hacer en un corpus de **NLTK** es estudiar el texto. La función `concordance` busca todas las ocurrencias de una palabra, es decir, cuantas veces se repite esa palabra en el texto junto con su contexto.

La primera vez que se ejecuta esta función, el sistema tardará unos pocos segundos en producir un resultado, posteriores ejecuciones serán más rápidas. En esa primera llamada se crea un índice con los contenidos para realizar búsquedas que, una vez creado, se usa en las llamadas siguientes para acelerar las búsquedas.

Primero hay que crear el corpus como un objeto `Text`. Luego se usa la función `concordance` sobre dicho objeto, pasándole como parámetro la palabra a buscar. Esta función ignora mayúsculas y minúsculas.



Ejemplo

```
alice_t = nltk.Text(alice)

alice_t.concordance('hatter', lines=3)

Output:

Displaying 3 of 56 matches:

    ing its right paw round , ' lives a Hatter : and in THAT direction ,
    waving t

    I almost wish I ' d gone to see the Hatter instead !' CHAPTER VII . A
    Mad Tea

    house , and the March Hare and the Hatter were having tea at it : a
    Dormouse
```

Buscando la palabra *hatter* en el texto resultan 56 ocurrencias de dicha palabra, y muestra el punto del texto donde aparecen. Pero esta función solo muestra información a la consola, no es idónea para manipular los datos, en su lugar se debería usar el método `concordance_list` que devuelve una lista con todas las ocurrencias.



Ejemplo

```
concordance_list = alice_t.concordance_list("hatter", lines=3)

for concordance in concordance_list:

    print(concordance.line)

Output:

ing its right paw round , ' lives a Hatter : and in THAT direction ,'
waving t

I almost wish I ' d gone to see the Hatter instead !' CHAPTER VII . A
Mad Tea

house , and the March Hare and the Hatter were having tea at it : a
Dormouse
```

Otra manera de buscar una palabra es a través del contexto, es decir, la palabra anterior y posterior a la requerida. Para ello se usa la función `common_contexts`.



Ejemplo

```
alice_t.common_contexts(['hatter'])

Output:

the_. the_, the_: the_was the_went the_said a_: the_instead the_were
the_opened the_grumbled the_ the_shook the_continued the_with
the_asked the_and the_; the_looked the_added
```

Se pueden ver los resultados de la búsqueda anterior, donde con el carácter `_` aparecería la palabra buscada y todos los contextos en los que aparece con la palabra anterior y posterior.

El concepto de sinónimo para **NLTK** es el de aquellas palabras que comparten contexto. Para buscar palabras que tienen el mismo contexto que la buscada, se usa la función `similar`.



Ejemplo

```
alice_t.similar('hatter')

Output:

duchess king dormouse mouse gryphon cat door caterpillar queen court
rabbit hall game footman knave other table garden dodo pigeon
```

En este caso, buscar sinónimos de *hatte** en este texto devuelve palabras que contextualmente tienen significados parecidos, es decir, posibles personajes de la obra.

8.4. Analizar la frecuencia de las palabras

Calcular la frecuencia de las palabras que contiene un texto es un análisis habitual de los textos. Tanto, que tiene su propia función, `FreqDist`, que recibe la lista de palabras del texto que se quiere analizar como parámetro. Se usa para obtener una distribución estadística de todas las palabras en el texto.



Ejemplo

```
fd = nltk.FreqDist(alice)

fd.most_common(10)

Output:

[(',', 1993), ('"', 1731), ('the', 1527), ('and', 802), ('.', 764),
('to', 725), ('a', 615), ('I', 543), ('it', 527), ('she', 509)]
```

Usando la función `most_common` se obtienen los *n* resultados más comunes, donde se muestra cada elemento junto al número de veces que se repite. Del resultado obtenido se puede observar que la mayoría de elementos comunes son elementos de puntuación, preposiciones, artículos, y esto sucede en cualquier lengua.

Para eliminar los signos de puntuación se puede usar la función `punctuation` del módulo `string`, que devuelve una cadena de caracteres con dichos signos de puntuación. Se crea un filtro en el cual se incluyen todas las palabras excepto aquellas que están incluidas en dicha cadena. Además, se usa la función `isalpha` de las cadenas de Python, que determina si en una cadena todos sus caracteres pertenecen al alfabeto.



Ejemplo

```
import string

punctuation = set(string.punctuation)

alice_filter_p = [word for word in alice if word.lower() not in punctuation and word.isalpha()]

fd = nltk.FreqDist(alice_filter_p)

fd.most_common(8)

Output:

[('the', 1527), ('and', 802), ('to', 725), ('a', 615), ('I', 543),
 ('it', 527), ('she', 509), ('of', 500)]
```

Ahora ya no aparecen signos de puntuación, pero se sigue observando que la mayor frecuencia se la llevan preposiciones, artículos, etc. Dado que estos elementos suelen tener poco significado en el análisis de texto es buena idea quitarlos, se llaman *stopwords*. Palabras con poco significado que deben ser filtradas. No hay una regla general para determinar si una palabra pertenece a ese grupo, pero **NLTK** facilita una lista de *stopwords* preseleccionada. Para cargar dicha lista, se usa el comando `download`.

```
nltk.download('stopwords')
```

Una vez descargada la lista de *stopwords* se seleccionan solo aquellas en el idioma deseado usando el método `words` visto anteriormente, y asignándole el resultado a una variable.



Ejemplo

```
sw_en = set(nltk.corpus.stopwords.words('english'))

print(len(sw_en))

list(sw_en)[:5]

Output:

179

['under', 'while', 'in', 'of', 'doing']
```

Existen 179 *stopwords* en el vocabulario inglés de acuerdo a **NLTK**. Se pueden usar dichas palabras para aumentar el filtro sobre el resultado de la distribución anterior.



Ejemplo

```
alice_filter_p = [word for word in alice if word.lower () not in sw_
en and word.lower () not in punctuation and word.isalpha ()]

fd = nltk.FreqDist(alice_filter_p)

fd.most_common(8)

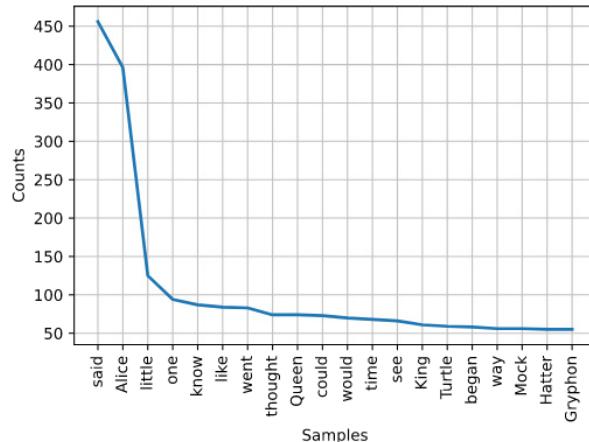
Output:

[('said', 456), ('Alice', 396), ('little', 125), ('one', 94), ('know',
87), ('like', 84), ('went', 83), ('thought', 74)]
```

Usando **Matplotlib** se puede obtener la gráfica de la distribución.

```
fd.plot(20, cumulative=False)
```

Programa 26. Realizar gráfica de distribución de palabras.



Salida 26. Resultado de la distribución de palabras.

8.5. Seleccionar palabras de un texto

Otra forma de procesamiento y análisis de datos es el proceso de seleccionar palabras en un texto basándose en unas características particulares. Por ejemplo, en su longitud. Para conseguir las palabras más largas de un número de caracteres se usa la función `len`.



Ejemplo

```
long_words = [word for word in alice if len(word) > 12]  
sorted(set(long_words))
```

Output:

```
['Multiplication', 'affectionately', 'circumstances', 'contemp-  
tuously', 'conversations', 'disappointment', 'extraordinary', 'inqui-  
sitively', 'straightening', 'uncomfortable', 'uncomfortably']
```

Se ordenan los resultados y se usa un set para evitar repeticiones, y se observa que hay 11 palabras que cumplen el criterio de tener más de 12 caracteres de longitud.

Otro ejemplo puede ser buscar todas las palabras que contengan una cierta secuencia de caracteres, como 'ious', que en inglés es un sufijo que se usa para decir que tiene las cualidades del significado de la raíz.



Ejemplo

```
ious_words = [word for word in alice if word.endswith('ious')]  
sorted(set(ious_words))
```

Output:

```
['anxious', 'curious', 'furious', 'various']
```

8.6. Bigramas y colocaciones

Otro elemento básico del análisis de texto es el uso de parejas de palabras (bigramas) en lugar de palabras simples. Las palabras 'es' y 'tarde' son un ejemplo de bigrama, puesto que su combinación es posible y tiene significado, con lo cual 'es tarde' se puede encontrar en datos textuales. Hay muchos de esos bigramas que son tan comunes en la literatura que casi siempre se usan juntos, como 'buenos días', 'entrar ganas', 'albergar esperanzas'. Estos bigramas se llaman colocaciones.

El análisis textual también puede suponer buscar bigramas en el texto bajo estudio, para encontrarlos existe la función `bigrams`.



Ejemplo

```
bigrams = nltk.FreqDist(nltk.bigrams(alice_filter_p))

bigrams.most_common(10)

Output:

[('said', 'Alice'), 123], ('Mock', 'Turtle'), 56], ('March',
'Hare'), 31], ('said', 'King'), 29], ('thought', 'Alice'), 26],
('White', 'Rabbit'), 22], ('said', 'Hatter'), 22], ('said', 'Mock'),
20], ('said', 'Caterpillar'), 18], ('said', 'Gryphon'), 18]]
```

Además de bigramas, también pueden encontrarse trigramas (combinaciones de tres palabras) con la función `trigrams`, o quadgrama (combinaciones de cuatro palabras) con `quadgrams`.



Ejemplo

```
trigrams = nltk.FreqDist(nltk.trigrams(alice_filter_p))

trigrams.most_common(10)

Output:

[('said', 'Mock', 'Turtle'), 20], ('said', 'March', 'Hare'), 10],
('little', 'golden', 'key'), 5], ('poor', 'little', 'thing'), 5],
('white', 'kid', 'gloves'), 5], ('March', 'Hare', 'said'), 5],
('Mock', 'Turtle', 'said'), 5], ('certainly', 'said', 'Alice'), 4],
('know', 'said', 'Alice'), 4], ('might', 'well', 'say'), 4]]
```

8.7. Sinónimos y antónimos

El paquete `wordnet` de **NLTK** incluye grupos de sinónimos de palabras y una breve definición. Se instala dicho paquete con `download`.

```
nltk.download("wordnet")
```

Se usa el método `synsets` de `wordnet` para obtener las definiciones y ejemplos de cualquier palabra.



Ejemplo

```
from nltk.corpus import wordnet

syn = wordnet.synsets("fun")

print(syn[0].definition())
print(syn[0].examples())

Output:

activities that are enjoyable or amusing

['I do it for the fun of it', 'he is fun to have around']
```

Y para obtener los posibles sinónimos de palabras se accede a la función `lemmas` del conjunto de sinónimos que devuelve la función `synsets`.



Ejemplo

```
synonyms = []

for syn in wordnet.synsets('sorrow'):

    for lemma in syn.lemmas():

        synonyms.append(lemma.name())

print(set(synonyms))

Output:

{'ruefulness', 'regret', 'grief', 'grieve', 'sorrowfulness',
'sadness', 'sorrow', 'rue'}
```

Para los antónimos simplemente hay que verificar si los valores que devuelve la función `lemmas` son `antonyms`.



Ejemplo

```
antonyms = []

for syn in wordnet.synsets("left"):

    for l in syn.lemmas():

        if l.antonyms():

            antonyms.append(l.antonyms()[0].name())

print(set(antonyms))

Output:

{'disinherit', 'center', 'right', 'enter', 'arrive'}
```

8.8. Stemming y lematización

Una técnica muy habitual en los motores de búsqueda al indexar páginas es buscar la raíz de las palabras, puesto que muchas personas escriben versiones diferentes de la misma palabra, y todas ellas derivan de la palabra raíz. De esta manera se normalizan dichas palabras y es más sencilla su búsqueda o tratamiento. Existen muchos algoritmos para esto, pero el más usado es el de Porter. Para esto, **NLTK** tiene una clase `PorterStemmer` que facilita el trabajo.



Ejemplo

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

print(stemmer.stem('deceases'))

Output:

Deceas
```

También dispone de `SnowballStemmer`, que soporta 13 idiomas además del inglés para esta funcionalidad.



Ejemplo

```
from nltk.stem import SnowballStemmer  
  
es_stemmer = SnowballStemmer('spanish')  
  
print(es_stemmer.stem('correr'))  
  
Output:  
  
Corr
```

En cambio, la lematización es el proceso de encontrar el lema de una palabra dependiendo de su significado y contexto. Normalmente se refiere al análisis morfológico de las palabras, cuyo objetivo es eliminar las terminaciones.

El algoritmo de *stemming* funciona quitando el sufijo de las palabras, es decir, corta su principio o final. El proceso de lematización es más potente, puesto que devuelve el lema de la palabra, que es la base para formar todas sus inflexiones, con lo que es una operación más inteligente.



Ejemplo

```
from nltk.stem import PorterStemmer  
  
stemmer = PorterStemmer()  
  
print(stemmer.stem('deceases'))  
  
Output:  
  
decease
```

Este proceso minimiza la ambigüedad del texto: las palabras se reducen y convierten todas aquellas con el mismo significado pero diferente representación a su forma más básica. Reduce la densidad de los textos y ayuda a prepararlos para un posterior entrenamiento.

8.9. Texto en internet

A menudo es interesante analizar texto directamente de una fuente online. Para ello se usa una librería como `urllib`, que permite la descarga de contenido de texto de internet, incluyendo páginas HTML.



Ejemplo

```
from urllib import request

url = "http://www.gutenberg.org/files/11/11-0.txt"

response = request.urlopen(url)

raw = response.read().decode('utf-8-sig')

raw[:81]

Output:

'The Project Gutenberg EBook of Alice's Adventures in Wonderland, by
Lewis Carroll'
```

La función `urlopen` de la librería `request` espera como parámetro la url de la página a descargar, en este caso un libro del Proyecto Gutenberg, y en formato crudo `raw`. Se lee y decodifica usando el set de caracteres correcto `utf-8-sig` y ya se puede acceder a su contenido.

Para trabajar con ello, hay que convertirlo a un corpus compatible con **NLTK**, es decir, hay que tokenizarlo, dividir el texto en palabras. Se realiza con la función `word_tokenize` y se convierten los *tokens* a un cuerpo de texto adecuado a **NLTK** con `Text`. A partir de ese momento ya se tiene un corpus valido para realizar cualquier análisis textual.



Ejemplo

```
tokens = nltk.word_tokenize (raw)

webtext = nltk.Text(tokens)

webtext[:11]

Output:

['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Alice', "'", 's',
'Adventures', 'in', 'Wonderland']
```

8.10. Extraer texto de páginas HTML

La mayor parte de la documentación en Internet está en formato de páginas HTML. Se suele usar la función `request` para descargarse el contenido HTML de una web.



Ejemplo

```
url = "https://www.universidadviu.com/es/la-universidad"  
  
html = request.urlopen(url).read().decode('utf-8-sig')  
  
html[:90]  
  
Output:  
  
'<!DOCTYPE html>\n<html lang="es-es" dir="ltr" prefix="og: https://  
ogp.me/ns#">\n <head>\n '
```

Ahora, hay que usar una librería adicional para convertir esa página HTML en un corpus de **NLTK**. **Beautiful Soup** es una librería especializada en analizar documentos HTML. Instalar **Beautiful Soup** es sencillo usando el gestor de paquetes *standard* de Python pip.

```
pip install beautifulsoup4
```

Esta librería ofrece analizadores sintácticos o *parsers* adecuados para reconocer etiquetas de código HTML y extraer el texto que las contiene y obtener un corpus válido con el que poder trabajar.



Ejemplo

```
from bs4 import BeautifulSoup  
  
raw = BeautifulSoup(html, "html.parser").get_text()  
  
tokens = nltk.word_tokenize(raw)  
  
text = nltk.Text(tokens)  
  
text[:4]  
  
Output:  
  
['La', 'Universidad', '|', 'VIU']
```

8.11. Análisis de sentimiento

El análisis de sentimiento es un nuevo campo de investigación que se ha desarrollado recientemente para poder evaluar la opinión de la gente acerca de un tema en particular. Se basa en diferentes técnicas que usan el análisis de texto en el mundo de las redes sociales o foros, a este se le llama minería de opiniones. Gracias a los comentarios y críticas de los usuarios, los algoritmos de análisis de sentimiento pueden evaluar el grado de apreciación o evaluación basándose en determinadas palabras clave. Este grado de apreciación se llama opinión, y suele tener tres valores posibles: positivo, neutral o negativo. La valoración de esta opinión se convierte de esta manera en una manera de clasificación.

Como ejemplo, se descarga el corpus de *movie_reviews* de NLTK, que contiene muchas críticas de películas, en las cuales hay texto junto con otro campo, que especifica si el resultado de la crítica es positivo o negativo. Se intentan encontrar las palabras más recurrentes en los documentos negativos o en los positivos para incidir en las palabras claves relacionadas a una opinión. Esto se puede realizar a través de un Clasificador Naive Bayes integrado en **NLTK**.

Para descargar el corpus se usa el método `download`.

```
nltk.download('movie_reviews')
```

Se construye el conjunto de entrenamiento a partir de dicho corpus, creando una lista de parejas de elementos que se llama `documents`. Esta lista contiene en el primer campo el texto de la crítica y en el segundo campo su evaluación negativa o positiva. Al final, se mezclan todos los elementos en orden aleatorio.



Ejemplo

```
import random

reviews = nltk.corpus.movie_reviews

documents = [(list(reviews.words(fileid)), category) for category in
reviews.categories() for fileid in reviews.fileids(category)]

random.shuffle(documents)
```

Se visualiza el primer elemento, primero su crítica con todas las palabras usadas y luego el segundo campo con la evaluación de dicha crítica.



Ejemplo

```
first_review = ' '.join(documents[0][0])
first_evaluation = documents[0][1]
print(f"Eval = {first_evaluation}, Review = {first_review}")

Output:
```

Eval = pos, Review = warning : contains what the matrix is . rated r
 for sci - fi violence . starring : keanu reeves , laurence fishburne ,
 joe panteliano some may be disappointed with the matrix . i ' ll
 tell you i was

Hay que crear la distribución de frecuencia de todas las palabras en el corpus, y se convierte a una lista.



Ejemplo

```
all_words = nltk.FreqDist(word.lower() for word in reviews.words())
word_features = list(all_words)
```

Se define una función para el cálculo de las *features*, las palabras que son suficientemente importantes para establecer la opinión de una crítica.



Ejemplo

```
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features ['{}'.format(word)] = (word in document_words)
    return features
```

En este momento ya se pueden crear conjuntos de *features* a partir de los documentos.



Ejemplo

```
featuresets = [(document_features(d, word_features), c) for (d, c) in documents]
```

El objetivo es crear un conjunto de todas las palabras contenidas en todo el corpus, analizar si están presentes en cada crítica individual, y averiguar cuánto contribuyen al juicio positivo o negativo. Si una palabra está presente más veces en las críticas negativas que en las positivas, se evalúa como una palabra mala. En el caso opuesto sería una palabra buena.

Para determinar cómo subdividir este conjunto de *features* para el conjunto de entrenamiento y de pruebas, primero hay que ver cuántos elementos contiene.

```
len(featuresets)
```

Si tiene 2000 elementos, como parece, se usan 1500 para el conjunto de entrenamiento, y los otros 500 para el conjunto de prueba para evaluar la precisión del modelo.



Ejemplo

```
train_set, test_set = featuresets[:1500], featuresets[1500:]
```

Finalmente, se aplica el Clasificador Naïve Bayes para clasificar el problema.



Ejemplo

```
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Después se calcula su precisión a través del conjunto de prueba.



Ejemplo

```
print(nltk.classify.accuracy(classifier, test_set))
```

Output:

0.802

Se observa que la precisión no es demasiado alta, pero hay que tener en cuenta que se está trabajando con palabras contenidas en un texto y, por tanto, es muy difícil crear modelos precisos relativos a problemas numéricos. Se puede ver que palabras tienen más peso a la hora de realizar una evaluación positiva o negativa de una crítica.



Ejemplo

```
classifier.show_most_informative_features(10)
```

Output:

Most Informative Features

| | | | |
|--------------------|-----------|---|------------|
| insulting = True | neg : pos | = | 15.9 : 1.0 |
| marvelous = True | pos : neg | = | 14.0 : 1.0 |
| sucks = True | neg : pos | = | 13.3 : 1.0 |
| outstanding = True | pos : neg | = | 11.5 : 1.0 |
| idiotic = True | neg : pos | = | 10.3 : 1.0 |
| ludicrous = True | neg : pos | = | 10.3 : 1.0 |
| fascination = True | pos : neg | = | 9.9 : 1.0 |
| filling = True | pos : neg | = | 9.9 : 1.0 |
| hatred = True | pos : neg | = | 9.9 : 1.0 |
| lean = True | pos : neg | = | 9.9 : 1.0 |

No resulta sorprendente que palabras como *sucks* o *insulting* sean consideradas negativas, mientras que *outstanding* o *marvelous* positivas.



Capítulo 9

Reconocimiento de dígitos escritos a mano con scikit-learn

El reconocimiento de texto escrito a mano es un problema que se remonta a las primeras máquinas automáticas que necesitaban reconocer caracteres individuales en documentos manuales. Ejemplo de ello son los códigos postales en las cartas en las oficinas postales y la automatización necesaria para reconocer los cinco dígitos. Un reconocimiento perfecto de esos códigos es necesario para ordenar y dirigir el correo postal de manera automática y eficiente.

En Python, la librería **scikit-learn** permite aproximarse a este tipo de análisis de datos, imágenes de números escritos a mano. Los datos a analizar están íntimamente relacionados con valores numéricos o cadenas de texto, pero también pueden provenir de imágenes y sonidos.

El problema consiste en predecir un valor numérico leyendo e interpretando una imagen que usa un tipo de letra escrito a mano. Se utilizará un estimador con la tarea de aprender, y una vez que haya llegado a un determinado grado de capacidad predictiva (un modelo lo suficientemente válido), producirá predicciones.

Se instala la librería **scikit-learn** a través del instalador de paquetes de Python, pip.

```
pip install scikit-learn
```

9.1. El conjunto de datos de dígitos

La librería **scikit-learn** contiene multitud de conjuntos de datos que son útiles para probar problemas de análisis de datos y para la predicción de los resultados. Existe un conjunto de datos de imágenes llamado Digits, que consiste en 1797 imágenes, cada una de 8x8 píxeles de tamaño. Cada imagen es un dígito escrito a mano en escala de grises.

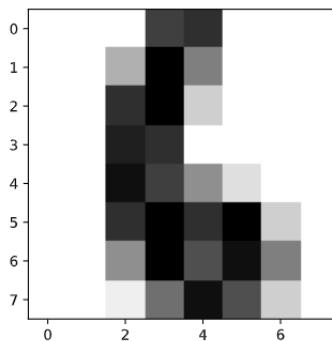


Figura 13. Imagen del dígito 6 escrito a mano.

Para cargar dicho conjunto de datos, se importan los datasets de **scikit-learn** y se llama a `load_digits`.

```
from sklearn import datasets

digits = datasets.load_digits()
```

Una vez cargado, se puede analizar el contenido; llamando al atributo `DESCR` de dicho conjunto de datos se accede a su información.

```
print(digits.DESCR)

Output:
```

Optical recognition of handwritten digits dataset

Data Set Characteristics:

```
:Number of Instances: 1797
:Number of Attributes: 64
:Attribute Information: 8x8 image of integer pixels in the range 0..16.
:Missing Attribute Values: None
:Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
:Date: July, 1998
```

The data set contains images of hand-written digits: 10 classes where each class refers to a digit.

Las imágenes de dichos dígitos se encuentran en un array, en la propiedad `images` dentro del conjunto de datos. Cada elemento dentro de este *array* es una imagen representada por una matriz de dimensiones 8x8 de valores numéricos que corresponden a un color de gris, entre el blanco (0) y el negro (15).



Ejemplo

```
digits.images[0]
```

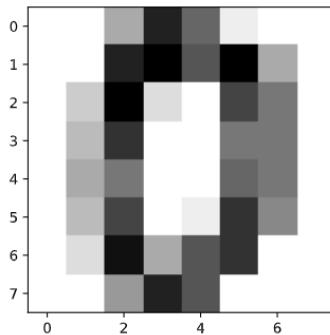
Output:

```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

Se puede visualizar dicha matriz usando la librería **Matplotlib**, con el método `imshow`. Se obtiene una imagen en escala de grises del dígito 0.

```
import matplotlib.pyplot as plt
plt.imshow(digits.images[6], cmap=plt.cm.gray_r, interpolation='nearest')
```

Programa 27. Ploteado de un dígito.



Salida 27. Resultado del pintado de un dígito.

Los valores numéricos que representan las imágenes (*targets*) están contenidos en el array `targets`, y verifican su longitud.

Ejemplo

```
print(digits.target)

print(digits.target.size)

Output:
```

[0 1 2 ... 8 9 8]

1797

9.2. Aprendiendo y prediciendo

Una vez cargados los dígitos, hay que definir un estimador SVC. Los SVC son máquinas de vectores de soporte (*support vector machines*), y representan un conjunto de algoritmos desarrollados para ayudar en escenarios de clasificación o regresión. En resumen, partiendo de un conjunto de datos en un plano, que pertenecen a dos clases distintas, estos algoritmos se encargan de encontrar una recta (hiperplano) que permita separar ambos conjuntos de datos.

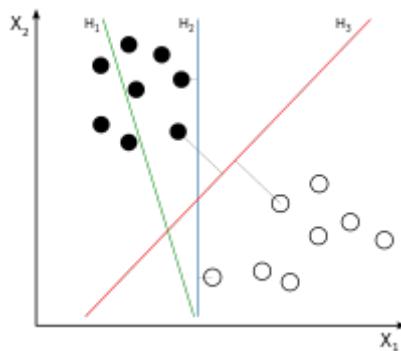


Figura 14. Separación de clases en un SVM.

Cuando se define un modelo predictivo, es necesario entrenarlo con un conjunto de entrenamiento, que es un conjunto de datos sobre los cuales ya se sabe a qué clase pertenecen. Dada la gran cantidad de elementos en el conjunto de datos de `Digits`, se podrá obtener un modelo muy efectivo, que sea capaz de reconocer con gran certeza los números escritos a mano.

Se crea el estimador SVC, con valores genéricos para C y gamma. Dichos valores se pueden ajustar de diferentes maneras a lo largo del análisis.



Ejemplo

```
from sklearn import svm  
  
svc = svm.SVC(gamma=0.001, C=100.)
```

Antes de entrenar el modelo hay que convertir las imágenes, que son matrices bidimensionales de 8x8 a una sola dimensión de 64. Con esto, el conjunto completo será de tamaño bidimensional (1797, 64), 1797 imágenes que tiene el conjunto de datos por 64 valores que tiene cada imagen ahora aplanada, donde cada uno de esos valores es el color en escala de grises de cada píxel.



Ejemplo

```
n_samples = len(digits.images)  
  
data = digits.images.reshape((n_samples, -1))
```

Suele ser necesario dividir los datos en dos secciones: una se usa para entrenar el modelo y la otra para validar. Para ello se utiliza la función `train_test_split` de **scikit-learn**. Se le pasa el conjunto de datos, junto con los valores que se deben predecir, en este caso en `target` del conjunto de datos de `digits`. Además, se le provee del porcentaje (expresado entre 0.0 y 1.0) de los datos que se van a usar para entrenar y de aquellos que se usarán para validar. En este caso, se usa un 5 % de los datos para validar y el otro 95 % para entrenar. Cuantos más datos haya disponibles para el entrenamiento, es razonable pensar que más precisión tendrá el modelo generado en sus validaciones. Esta función devuelve las matrices divididas en cuatro; en `X` se tienen los conjuntos de datos para entrenamiento y validación, mientras que en `y` se tienen los valores también para entrenamiento y validación, pero con los resultados que se deben predecir.



Ejemplo

```
X_train, X_test, y_train, y_test = train_test_split(data, digits.  
target, test_size=0.05, shuffle=False)
```

Por último, queda indicarle al SVC que empiece a aprender a partir del conjunto de datos de entrenamiento y los resultados a predecir.



Ejemplo

```
svc.fit(X_train, y_train)
```

Ahora que ya se ha entrenado al estimador, se intentan predecir los resultados para el resto del conjunto de datos, llamando a la función `predict` con el conjunto de datos de validación, pero sin pasarle los resultados, que luego se compararán para estimar la precisión del modelo.

Ejemplo

```
predicted = svc.predict(X_test)

Predicted

Output:

array([ 8,  4,  1,  7,  7,  3,  5,  1,  0,  0,  2,  2,  7,  8,  2,  0,
       1,  2,  6,  3,  8,  7,  5,  3,  4,  6,  6,  6,  4,  9,  1,  5,  0,  9,  5,  2,  8,  2,  0,  0,  1,  7,  6,  3,
       2,  1,  7,  4,  6,  3,  1,  3,  9,  1,  7,  6,  8,  4,  3,  1,  4,  0,  5,  3,  6,  9,
       6,  1,  7,  5,  4,  4,  7,  2,  8,  2,  2,  5,  7,  9,  5,  4,  8,  8,  4,  9,  0,  8,
       9,  8])
```

El resultado es la predicción del modelo para el conjunto de validación, pero se utilizará **Matplotlib** para visualizar la imagen que tiene que predecir y junto a ella el valor que ha predicho el modelo.

```
_, axes = plt.subplots(nrows=1, ncols=6, figsize=(10, 3))

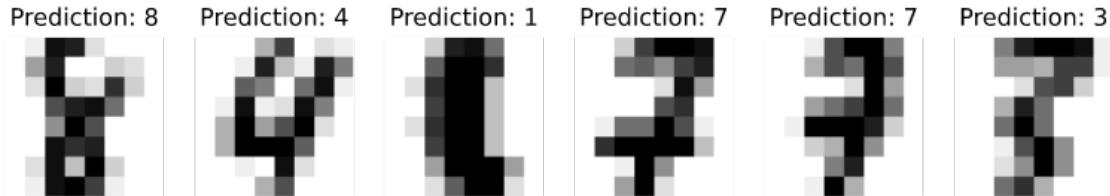
for ax, image, prediction in zip(axes, X_test, predicted):
    ax.set_axis_off()

    image = image.reshape(8, 8)

    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')

    ax.set_title(f'Prediction: {prediction}')
```

Programa 28. Pintado de 6 dígitos con sus respectivas predicciones.



Salida 28. Resultado de las predicciones.

Se crea un gráfico de 6 columnas, y se recorre en un `zip` para cada una de esas columnas el conjunto de validación, junto con el valor predicho por el modelo. Para cada una de las columnas, se redimensiona la imagen de vuelta a una matriz cuadrada de 8x8 para deshacer el aplanamiento realizado anteriormente, necesario para el entrenamiento, y devolverle a la imagen su formato original. A continuación, se muestra la imagen y se pone como título de esta el valor predicho para verificar visualmente que se corresponden.

La librería **scikit_learn** ofrece la posibilidad de generar un pequeño informe sobre el resultado del entrenamiento y la predicción. Se usa `classification_report` y se pasan como parámetros los valores de los resultados esperados para el conjunto de validación y los que ha predicho el modelo para compararlos.



Ejemplo

```
from sklearn import metrics

print(metrics.classification_report(y_test, predicted))
```

Output:

| | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 1.00 | 1.00 | 10 |
| 2 | 1.00 | 1.00 | 1.00 | 10 |
| 3 | 1.00 | 0.80 | 0.89 | 10 |
| 4 | 1.00 | 1.00 | 1.00 | 10 |
| 5 | 0.88 | 1.00 | 0.93 | 7 |
| 6 | 1.00 | 1.00 | 1.00 | 9 |
| 7 | 1.00 | 1.00 | 1.00 | 10 |
| 8 | 0.90 | 1.00 | 0.95 | 9 |
| 9 | 1.00 | 1.00 | 1.00 | 7 |
| accuracy | | | 0.98 | 90 |
| macro avg | 0.98 | 0.98 | 0.98 | 90 |
| weighted avg | 0.98 | 0.98 | 0.98 | 90 |

Se puede observar que el valor de precisión para el modelo que se ha creado está muy cerca de 1.00 y, por tanto, es capaz de interpretar y predecir dichos dígitos escritos a mano con una gran precisión.



Capítulo 10

Análisis de imágenes y reconocimiento de caras con OpenCV

Esta rama del análisis está basada en varias técnicas de cálculo aplicadas a las imágenes; estas técnicas se llaman filtros de imágenes. A partir del desarrollo del *deep learning*, el análisis de imágenes ha experimentado un gran empuje hasta llegar a resolver problemas que anteriormente eran imposibles, lo que ha dado lugar a una nueva disciplina llamada *computer vision*.

La visión por ordenador se considera una parte de la inteligencia artificial, ya que su propósito es reproducir la manera en la que el cerebro humano percibe las imágenes. De hecho, ver no consiste solo en la adquisición de una imagen bidimensional, sino, sobre todo, en la interpretación del contenido de dicha área. La imagen capturada se descompone y elabora en niveles de representación que son cada vez más abstractos (contornos, figuras, objetos, palabras, etc.) y, por tanto, más reconocibles por la mente humana. Del mismo modo, la visión por ordenador intenta procesar las imágenes y extraer los mismos niveles de representación. Esto se efectúa a través de varias operaciones:

- Detección: detectar formas, objetos u otros sujetos de investigación en una imagen; por ejemplo, encontrar coches.
- Reconocimiento: los sujetos identificados se agrupan en clases genéricas; por ejemplo, subdividir los coches por marcas o tipos.
- Identificación: se identifica una instancia de la clase previa; por ejemplo, encontrar mi coche.

10.1. OpenCV

OpenCV (Open Source Computer Vision) es una librería escrita en C++ que está especializada en visión por ordenador y análisis de imágenes. Soporta gran cantidad de algoritmos relacionados con la visión artificial y el *machine learning*, y está creciendo continuamente.

Para instalar **OpenCV** dentro de un entorno virtual de Python, como se recomienda, se hace uso del instalador de paquetes pip, y en concreto del paquete `opencv-contrib-python`, que es el repositorio no oficial que contiene la última versión de **OpenCV**, tanto los módulos principales como los opcionales.

```
pip install opencv-contrib-python
```

Si se quiere instalar en el sistema, en entornos Linux lo más sencillo es usar el gestor de paquetes `apt-get`, y para Windows suele haber disponibles unos archivos binarios precompilados que hay que instalar.

```
sudo apt-get install python3-opencv
```

Para verificar que **OpenCV** está instalado, se importa y se intenta imprimir la versión de la que se trate.

```
import cv2
print(cv2.__version__)
```

10.2. Cargar y mostrar una imagen

Puesto que **OpenCV** trabaja con imágenes, es importante saber cómo cargarlas, manipularlas y finalmente mostrarlas para ver los resultados.



Enlace de interés

Foto de un grupo de gente.

<https://www.pexels.com/photo/multi-cultural-people-3184419/>

Para leer un fichero que contiene una imagen se usa el método `imread`, que recibe como parámetro la ruta de la imagen en disco y la carga de un fichero comprimido como jpg, y la traduce a una estructura de datos compuesta por una matriz numérica que corresponde a posiciones e intensidades de color.

```
img = cv2.imread('people.jpg')
```

Si se accede a la imagen cargada con los suscriptores como si fuera una matriz bidimensional, se obtienen los tres componentes de color RGB para ese píxel en particular.

```
img[0,0]
```

Output:

```
array([156, 168, 178], dtype=uint8)
```

Visualizar una imagen es una tarea habitual, y se puede realizar llamando al método `imshow`, que crea una ventana. En el primer parámetro recibe el nombre de la ventana a crear, y en el segundo la variable con la imagen. Una vez creada la ventana, se puede usar la función `waitKey`, que espera el número de milisegundos indicado en el parámetro; si se le pasa un 0, espera hasta que se pulse una tecla. También es útil poder cerrar dicha ventana desde código con `destroyWindow`, que recibe como argumento el nombre de la ventana a cerrar. Si hay múltiples ventanas abiertas y se quieren cerrar todas a la vez, existe el método `destroyAllWindows`.

```
cv2.imshow('Image', img)
```

```
cv2.waitKey(0)
```

```
cv2.destroyAllWindows()
```

Programa 29. Visualización de una imagen con OpenCV.



Salida 29. Resultado de visualización.

10.3. Trabajando con imágenes

Una imagen es una matriz bidimensional, cada uno de cuyos elementos representa un punto de esta llamado píxel. Cada píxel suele ir codificado con su color, que suele estar formado por 3 componentes en el formato RGB: rojo, verde y azul. Cada uno de estos componentes puede tener un valor entre 0 y 255, siendo 0 la ausencia de dicho componente en el color final, y 255 la contribución máxima. De esta manera, los colores se componen sumando las intensidades de sus tres componentes o colores básicos, rojo, verde y azul.

Una vez cargada la imagen, se puede descomponer en los tres canales RGB usando el método `split`. Es necesario tener en cuenta que **OpenCV** almacena los canales por defecto en el orden BGR.

```
b, g, r = cv2.split(img)
```

Para volver a componer los tres canales se usa el método `merge`, pero si se intercambia uno de los canales, se puede observar fácilmente el efecto de alterar los colores en la imagen final.

```
img_alt = cv2.merge((b,r,g))
```

Una vez manipulada la imagen, se puede guardar en el sistema de archivos con el método `imwrite`, que recibe como primer parámetro el nombre del fichero a guardar, y como segundo la imagen. El formato del fichero de imagen vendrá determinado por la extensión del archivo.

```
cv2.imwrite("people_alt.jpg", img_alt)
```



Figura 15. Imagen con los canales verde y rojo intercambiados.

10.4. Operaciones elementales con imágenes

Una de las operaciones más básicas es la suma de dos imágenes. Con **OpenCV** es sencilla de ejecutar usando el método `add`. Este recibe como parámetros dos imágenes y devuelve una tercera con el resultado de haber combinado ambas. Las dos imágenes a combinar deben tener el mismo tamaño.



Enlace de interés

Enlace de la segunda foto usada para mostrar el blending.

<https://www.pexels.com/photo/city-fashion-people-woman-7298990/>

```
img_color = cv2.imread('color.jpg')
```

```
img_add = cv2.add(img, img_color)
```



Figura 16. Dos imágenes mezcladas haciendo uso del método `add`.

El resultado quizás no es el esperado; la prevalencia del blanco se debe a la simple suma aritmética de los tres valores RGB, que se calcula para cada píxel individual. Cada componente de color puede tener como máximo un valor de 255, por lo que si la suma de los componentes de color de las dos imágenes es mayor a 255, este valor nunca sobrepasa esa cifra. Esto provoca que la suma de dos imágenes no dé como resultado una sencilla mezcla de ambas, sino que se tienda hacia el blanco.

Se puede realizar también la operación de resta de dos imágenes con el método `subtract`; en este caso, lo esperado es lo contrario, una imagen que tienda hacia el color negro.

```
img_sub = cv2.subtract(img, img_color)
```



Figura 17. Dos imágenes restadas.

Puesto que la resta no es commutativa, el resultado depende del orden en el que se realice dicha resta. La librería **OpenCV** se apoya en **numPy** en sus estructuras internas y, por tanto, cada objeto de imagen creado no es más que un array de array* de **numPy**. Por ello, se pueden usar las operaciones de matrices que provee **numPy**, como por ejemplo la suma de matrices.

```
img_add = img + img_color
```



Figura 18. Dos imágenes sumadas usando numPy.

Pero es preciso tener en cuenta que el resultado no será el mismo. Las funciones `add` y `subtract` mantienen los valores de los componentes de color entre 0 y 255, independientemente del valor de los operadores. Usando la suma directa con **numPy**, si la suma del componente de un color excede 255, el resultado se interpreta de manera distinta y se crea un efecto extraño, probablemente como el módulo de 255. Para la resta sucede un efecto similar: pueden darse valores negativos por debajo de 0. En caso de usar las funciones de **OpenCV**, todos los valores tendrán 0 como mínimo.

10.5. Blending

La suma o resta de dos imágenes no produce una imagen intermedia entre ambas, sino que las colorea hacia el blanco o el negro. La operación correcta para esta operación es el *blending*. Se puede considerar como la superposición de una imagen sobre la otra. Ajustando gradualmente la transparencia de la primera se obtiene una mezcla nueva e intermedia de las dos.

La operación de *blending* no se corresponde con una simple suma, sino con la siguiente ecuación:

```
img = alpha . img1 + (1 - alpha) . img2
```

donde $0 \geq \alpha \geq 1$

Las dos imágenes tienen dos coeficientes numéricos que toman valores entre 0 y 1. Cuando el valor del parámetro *alpha* crece, se produce una transición suave de la primera imagen hacia la segunda. En **OpenCV** existe el método `addWeighted` para realizar el *blending* de esta manera. Esta función recibe las dos imágenes a mezclar, los dos valores de transparencia para cada imagen, y un último valor *gamma* que se sumará siempre al resultado final, siguiendo esta ecuación:

```
dst = src1 * alpha + src2 * beta + gamma  
img_blend = cv2.addWeighted(img, 0.7, img_color, 0.3, 0)
```

Lo cual produce el resultado deseado de mezclar ambas imágenes correctamente.



Figura 19. Dos imágenes usando un *blending* correcto.

10.6. Detección de bordes

Ya se ha visto que las imágenes no son otra cosa que arrays de **numPy**; por tanto, estas matrices numéricas pueden ser procesadas, y se pueden implementar funciones matemáticas que procesen los números de dichas matrices para obtener nuevas imágenes. Estas nuevas imágenes que se obtengan de las operaciones servirán, a su vez, para proveer nueva información. Este es el concepto detrás del análisis de imágenes. Las operaciones matemáticas necesarias para llegar de una imagen original hasta una imagen resultante se llaman filtros de imagen.

Una de las operaciones fundamentales al analizar una imagen es entender el contenido de esta, como objetos o personas. Pero para hacerlo, primero hay que entender las posibles formas que están representadas en la imagen. Y para entender la geometría representada, es necesario reconocer los contornos que delimitan un objeto del fondo o de otros objetos. Esto es la detección de bordes.

10.6.1. Teoría del gradiente de imagen

Entre las muchas operaciones que se pueden realizar con las imágenes se encuentran las convoluciones, mediante las cuales se aplican ciertos filtros para editarlas con el fin de obtener información o cualquier otra utilidad. Las convoluciones procesan los valores numéricos de la imagen a través de operaciones matemáticas (filtros de imagen) para producir nuevos valores en una nueva matriz del mismo tamaño.

Una de estas operaciones es la derivada, que en términos matemáticos permite obtener valores numéricos que indican la velocidad a la que un valor cambia. En relación con la variación del color en las imágenes, esto recibe el nombre de gradiente.

Ser capaz de calcular el gradiente de un color es una buena herramienta para determinar los bordes de una imagen; de hecho, el ojo humano puede distinguir los bordes de una figura dentro de una imagen gracias a los saltos entre un color y otro. Además, el ojo puede percibir la profundidad gracias a las diferentes tonalidades de color desde el claro al oscuro, el gradiente. Así pues, medir el gradiente de una imagen es crucial para detectar los bordes de esta. Se lleva a cabo con un simple filtro que se ejecuta sobre una imagen.

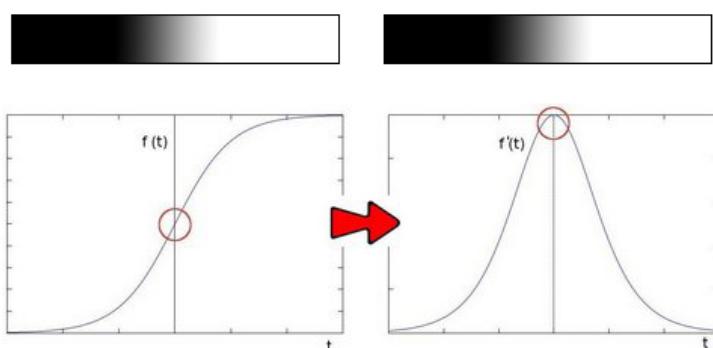


Figura 20. Gradiente de negro a blanco con su función y su derivada.

En la parte superior de la figura anterior se muestra una imagen que tiene un gradiente de negro a blanco. Un borde no es más que una transición rápida de un tono a otro. El 0 es negro y el 1 es blanco, y los tonos de gris son valores en coma flotante entre 0 y 1. Si se dibujan todos los valores de los colores del gradiente en un gráfico, se obtiene la función $f()$. Se puede observar una transición brusca de 0 a 1, lo que indica el borde.

La derivada de la función $f()$ resulta en la función f' . La máxima variación del tono lleva a valores cercanos a 1 a esta función. Así que cuando se convierten colores, se obtiene un gráfico donde el blanco indica un borde.



Enlace de interés

Imagen de prueba que tiene un contraste de color muy alto, y donde se pueden observar fácilmente todas las posibles orientaciones de algunos bordes (horizontales, verticales y diagonales).

<https://www.meccanismocomplesso.org/wp-content/uploads/2016/12/blackandwhite.jpg>

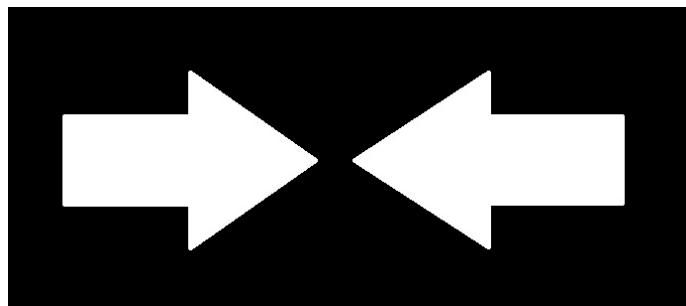


Figura 21. Imagen de prueba para detección de bordes.

En **OpenCV** hay dos filtros de imagen, Sobel y Laplacian, que corresponden a operaciones matemáticas matriciales. Se carga la imagen de prueba, se aplica la función `Laplacian`, la cual recibe como parámetros la imagen a analizar y la profundidad de color de la imagen, y, puesto que está en escala de grises, se usa el formato correcto `CV_64F`. Posteriormente se usa la función `Sobel`, que recibe la imagen, la profundidad, los órdenes de la derivada en las direcciones x e y, y por último el tamaño del *kernel* (debe ser 1, 3, 5 o 7 y representa el área donde se aplica el filtro para cada píxel). Se usa el filtro Sobel en la dirección horizontal y en la vertical, para poder observar las diferencias. A nivel de llamada al filtro, solo cambian los órdenes de las derivadas en los dos ejes.

Por último, se hace uso de **Matplotlib** para visualizar las imágenes con los filtros aplicados.

```
import matplotlib.pyplot as plt

img = cv2.imread('arrows.jpg', 0)

laplacian = cv2.Laplacian(img, cv2.CV_64F)

sobel_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)

sobel_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)

def show_image(sub, image, title):

    plt.subplot(2, 2, sub)

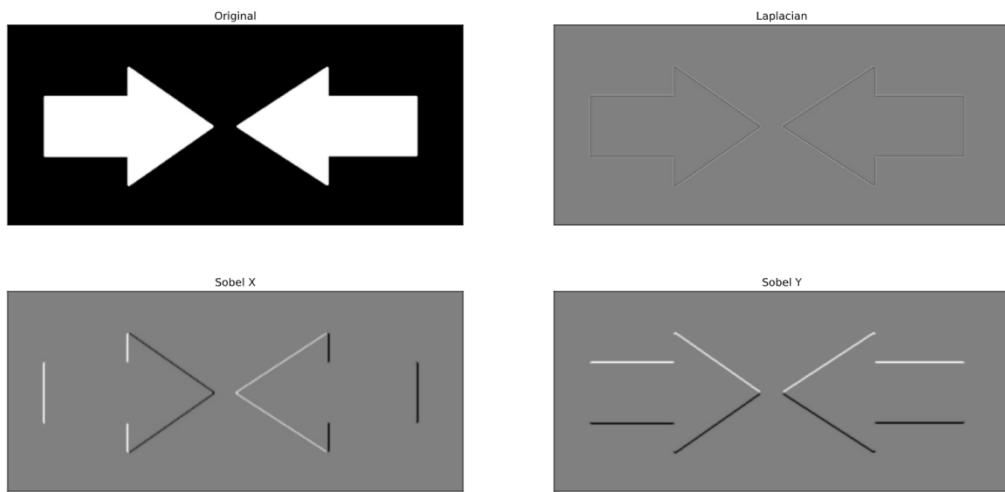
    plt.imshow(image, cmap = 'gray')

    plt.title(title)

    plt.xticks([])

    plt.yticks([])
```

```
show_image(1, img, 'Original')
show_image(2, laplacian, 'Laplacian')
show_image(3, sobel_x, 'Sobel X')
show_image(4, sobel_y, 'Sobel Y')
plt.show()
```

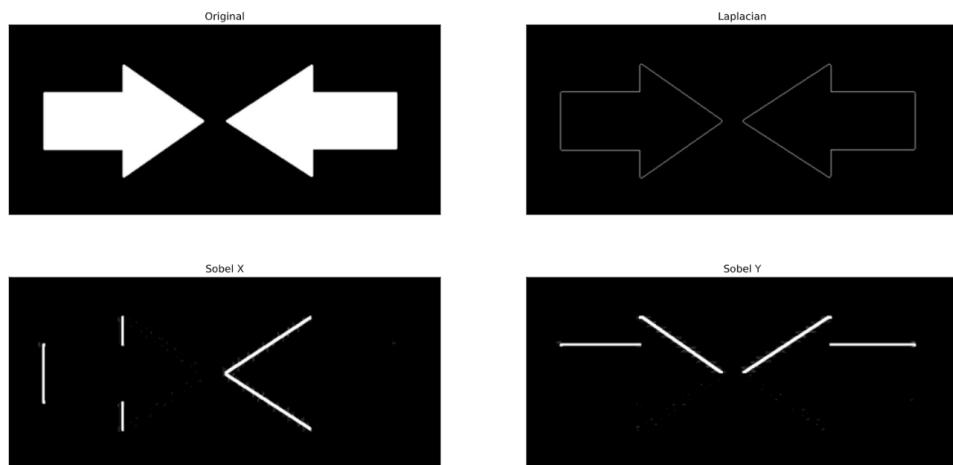
Programa 30. Generación de filtros Sobel y Laplacian.**Salida 30.** El resultado es una imagen con las cuatro representaciones de la imagen de prueba de las flechas.

Se puede apreciar que los filtros Sobel han realizado la detección de bordes a la perfección, incluso en los casos en que estaban limitados a horizontales o verticales. Las líneas diagonales son visibles en ambos casos, puesto que tienen componentes horizontales y verticales, pero los bordes horizontales en el filtro Sobel X y los verticales en Sobel Y no se detectan.

Al combinar los dos filtros (el cálculo de las dos derivadas) para obtener el filtro Laplacian, los bordes se determinan de manera omnidireccional, pero se pierde bastante resolución. Se pueden observar las ondulaciones de los bordes más tenues. Colorear en gris es útil para detectar bordes y gradientes, pero si interesa solo detectar bordes, entonces es mejor idea utilizar el formato CV_8U para usar 8 bits por píxel.

```
laplacian = cv2.Laplacian(img, cv2.CV_8U)
sobel_x = cv2.Sobel(img, cv2.CV_8U, 1, 0, ksize=5)
sobel_y = cv2.Sobel(img, cv2.CV_8U, 0, 1, ksize=5)
```

Programa 31. Generación de filtros Sobel y Laplacian con 8 bits.



Salida 31. Se observan resultados similares, pero esta vez los bordes se ven en blanco sobre fondo negro. No obstante, en ambos filtros Sobel han desaparecido los bordes que en la imagen en escala de grises aparecían en negro.

Esto se debe a un problema al convertir los datos. Los gradientes en la imagen en escala de grises con valores en el formato CV_64F están representados por valores positivos (inclinaciones positivas) al cambiar de negro a blanco. Sin embargo, cuando cambian de blanco a negro se representan con valores negativos (inclinaciones negativas). En la conversión de CV_64F a CV_8U, todas las pendientes negativas se reducen a 0, y así la información relativa esos bordes se pierde, por lo que cuando se muestra la imagen, los bordes de blanco a negro no se aprecian. Para solucionar esto, habría que guardar los datos en la salida del filtro en el formato CV_64F, calcular el valor absoluto y finalmente hacer la conversión a CV_8U.

```
import numpy as np

laplacian = cv2.Laplacian(img, cv2.CV_64F)

laplacian = np.uint8(np.absolute(laplacian))

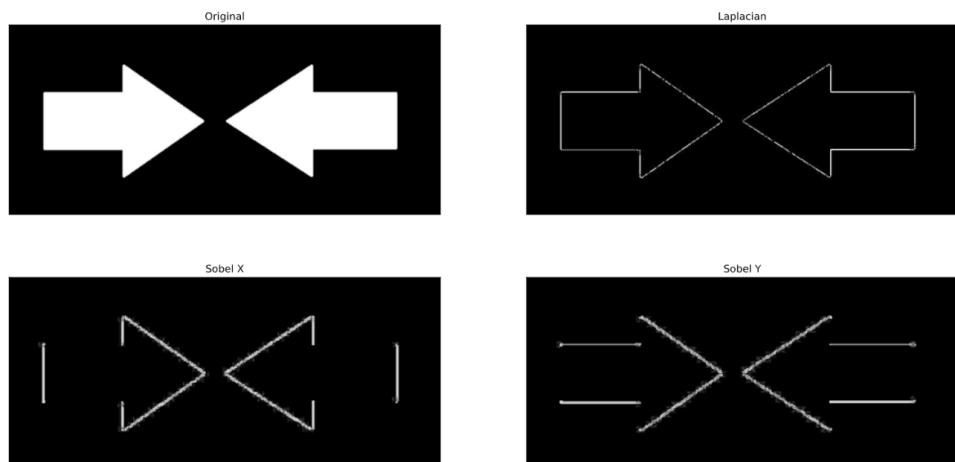
sobel_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)

sobel_x = np.uint8(np.absolute(sobel_x))

sobel_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)

sobel_y = np.uint8(np.absolute(sobel_y))
```

Programa 32. Generación de filtros Sobel y Laplacian con 8 bits y conversión.



Salida 32. Ahora se puede apreciar la representación correcta de los bordes en blanco sobre negro. Los únicos bordes que no se aprecian en los filtros Sobel son los paralelos a la dirección de la detección (horizontales y verticales).

Además de los bordes, estos filtros también son capaces de detectar el nivel de los gradientes mediante el uso de una escala de grises. Al utilizar como sujeto de pruebas otra imagen con multitud de gradientes horizontales y verticales, se puede observar cómo el filtro Laplacian aplicado sobre ella muestra los bordes con un escalado en función del nivel del cambio de color en los gradientes.

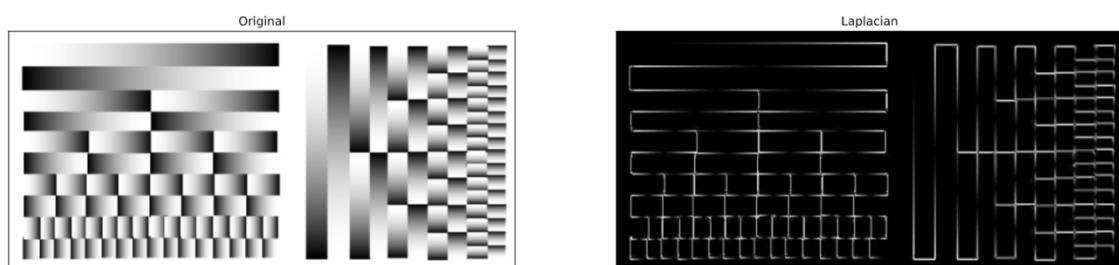


Figura 22. Filtro Laplacian sobre una imagen de gradientes.

10.7. Detección de caras

El proceso para detectar caras humanas en una imagen es bastante más complejo que el de detección de bordes. Dada la complejidad del problema, se usan técnicas de *deep learning*. De hecho, en la base de esta técnica se hallan redes neuronales especialmente diseñadas para reconocer diferentes cosas, entre ellas caras de personas, en una foto. La detección de objetos funciona de manera muy parecida.

Por simplicidad, se utilizará una red neuronal ya entrenada. Entrenar a una para este tipo de problema puede ser una operación de gran dificultad y requerir una cantidad significativa de tiempo y recursos. Afortunadamente, **OpenCV** cuenta con algunas redes neuronales ya entrenadas para realizar este tipo de operaciones. Se usará como ejemplo un modelo desarrollado mediante el framework **Caffe2**.

Para utilizar una red neuronal con modelos de **Caffe2** dentro de **OpenCV** se necesitan dos tipos de ficheros.



Enlace de interés

Fichero prototxt, que define la arquitectura del modelo.

https://github.com/opencv/opencv/blob/master/samples/dnn/face_detector/deploy.prototxt



Enlace de interés

Fichero con el modelo caffemodel, que contiene los pesos de las capas en la red neuronal. Este fichero es el más importante, puesto que alberga todo el aprendizaje de dicha red neuronal para realizar el trabajo. Se va a usar el siguiente modelo:

https://github.com/opencv/opencv_3rdparty/tree/dnn_samples_face_detector_20170830

La librería **OpenCV** soporta distintos frameworks de *deep learning* y tiene muchas características de soporte. En particular, cuenta con el módulo dnn, que se especializa en *deep neuronal networks*. Para cargar una red entrenada de **Caffe2** se usa la función `readNetFromCaffe`, donde se indican los dos ficheros anteriormente mencionados.



Ejemplo

```
net = cv2.dnn.readNetFromCaffe('deploy.prototxt.txt', 'res10_300x300_
ssd_iter_140000 caffemodel')
```

Como prueba se utilizará la foto de ejemplos anteriores en la que aparecían varias personas. Se consultan las dimensiones de la imagen en altura y anchura.



Ejemplo

```
img = cv2.imread('people.jpg')
(h, w) = img.shape[:2]
```

Otra función, `blobFromImage`, es la que se encarga del preprocesamiento de la imagen para adaptarla al trabajo con redes neuronales. Se redimensiona a un tamaño de 300x300 para que pueda ser usada por el modelo, que está entrenado para imágenes de ese tamaño. Dicha función toma como parámetros la imagen a analizar, un factor de escala, las dimensiones de la imagen y, por último, los valores medios para cada canal de color.



Ejemplo

```
blob = cv2.dnn.blobFromImage(cv2.resize(img, (300, 300)), 1.0, (300, 300), (104.0, 177.0, 123.0))
```

Se realiza la detección pasándole primero la imagen preprocesada a la red neuronal.



Ejemplo

```
net.setInput(blob)  
detections = net.forward()
```

La red devolverá en la variable `detections` un array de **NumPy** con todas las detecciones de caras que ha efectuado, y a cada una de esas detecciones le dará un valor de confianza. Se define un umbral de confianza a partir del cual se va a considerar que la detección es buena.



Ejemplo

```
confidence_threshold = 0.6
```

Por último, se recorre dicho array de detecciones y para cada una de ellas se comprueba si se rebasa el umbral de confianza. Si es afirmativo, entonces se consigue el rectángulo donde se ha efectuado dicha detección y se escala de vuelta al tamaño original de la imagen. Se crea un texto con el valor de confianza, y se pinta un rectángulo y ese texto en la imagen.



Ejemplo

```
for i in range(0, detections.shape[2]):  
  
    confidence = detections[0, 0, i, 2]  
  
    if confidence > confidence_threshold:  
  
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])  
  
        startX, startY, endX, endY = box.astype("int")  
  
        text = f"{confidence * 100:.2f}"  
  
        y = startY - 10 if startY - 10 > 10 else startY + 10  
  
        cv2.rectangle(img, (startX, startY), (endX, endY), (0, 0, 255), 2)  
  
        cv2.putText(img, text, (startX, y), cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 2)
```

Al ejecutar el código se genera la imagen con los resultados del procesamiento de la detección de caras. Todos los rostros de las personas en la imagen han sido detectados con éxito, por lo que se pueden observar los rectángulos sobre las caras y el porcentaje de confianza de cada detección, todos con valores muy altos.



Figura 23. Imagen con las caras reconocidas en los recuadros con su nivel de confianza.

Glosario

Broadcasting

Se trata de una operación interna de Python por la cual es posible realizar operaciones entre elementos de dimensiones distintas. En su forma más simple resulta en incrementar virtualmente el tamaño de la ndarray menor hasta igualar el de la otra para así poder aplicar la operación correctamente. Esto es lo que sucede en el caso básico de aplicar una operación entre un escalar y una ndarray (por ejemplo, `np.arange(10) * 2`).

Casting

Del inglés *cast*, casting es la operación por la cual se fuerza la conversión de un tipo de elemento a otro, como `int(1.1)` o `string(10)`.

Comparación condicional

Una comparación condicional es una expresión en código que se evalúa hasta dar con un elemento booleano (`True` o `False`). Las comparaciones entre dos objetos emplean comparadores como `==`, `is`, `!=`, `not`, `<`, `<=`, `>`, `>=` para determinar el valor final de la expresión. Se pueden combinar comparaciones con el uso de operadores lógicos como `&` (and), `|` (or) y `~` (not). Las comparaciones condicionales, combinadas con los comandos `if` y `else`, se utilizan para ramificar o bifurcar el código dependiendo del valor de una expresión.

Datos tabulares

Los datos formulados en dos dimensiones (filas y columnas) se denominan datos tabulares.

Debugging

Del inglés *debug*, se utiliza tradicionalmente en informática para hacer referencia al proceso por el cual el programador detecta, investiga y resuelve problemas (*bugs*) en su código.

Decorador

En **Matplotlib**, los decoradores son funciones que sirven para modificar el aspecto general de una figura y sus ejes.

Diccionario

Un diccionario es una colección de claves (índices) asociados a valores. Toma la forma `{ clave1 : valor1, clave2 : valor2, ... , claveN : valorN }`. Cada índice puede aparecer un máximo de una vez en el diccionario.

Función lambda

Es un tipo especial de función con la particularidad de que es anónima y, por lo tanto, solo se utiliza en el momento de declararla. En Python es muy frecuente expresar con una función lambda argumentos de métodos que aceptan otra función.

Función

Una función es un fragmento de código con *input* y *output* (ambos pueden ser nulos) que puede invocarse en otro punto del código. Se suelen usar para aglutinar funcionalidad de repetida utilidad (y evitar la repetición de código) y para separar código por funcionalidad.

Índice semántico

En **Pandas**, el índice semántico hace referencia al índice nominal o etiqueta (*label*) que posee cada uno de los elementos de una serie o *frame*. Se diferencia del índice posicional en que este último es siempre un número entero que describe la posición en la colección, mientras que el índice semántico puede ser un número o *string*.

Lenguaje interpretado

Un lenguaje se denomina interpretado cuando su ejecución se hace a partir de la lectura e interpretación concurrente del código escrito por el usuario. La lectura del código y su ejecución por parte de la computadora suceden de forma simultánea. Se distinguen de los lenguajes compilados en que estos requieren un paso previo (compilación) en el que el código se traduce a lenguaje máquina.

Limpieza de datos

Todo método que sirve para transformar, manipular o modificar los datos para su utilización posterior en técnicas de análisis de datos o aprendizaje automático.

List comprehension

Se trata de un concepto intrínseco a Python por el que se pueden formar colecciones a partir de expresiones con *loops* y condicionales.

Lista

Una lista es una colección de elementos. A diferencia de las tuplas, una lista es mutable, es decir, su contenido puede ser alterado una vez creada. Toma la forma `[a, b, c]`.

Loop

Un *loop* (o ciclo) es un bloque de código que puede evaluarse múltiples veces por repetición. En Python existen dos tipos de *loops*: `for` y `while`.

Método

Véase “función”.

Namespace

En lenguajes de programación, un *namespace* (espacio de la nomenclatura) es el ámbito en el cual se reconoce un determinado objeto, función o variable. En Python, hace referencia al lugar en el que se encuentra definida e implementada una clase o función, por lo que para su utilización se requiere importación.

Operador binario

Un operador binario es un operador que requiere dos operandos para formar un resultado. Ejemplos de operadores binarios son la adición, la resta y la multiplicación.

Operador ternario

Un operador ternario es una forma corta de una expresión `if else`, que combina la comparación condicional y las expresiones dependientes en una sola expresión. En Python toma la siguiente forma: <expresión1> `if` <comparación> `else` <expresión2>

Operador unario

Un operador unario es un operador que requiere de un solo operando para formar un resultado. Ejemplos de operadores unarios son el coseno, el seno y el logaritmo.

Script

En su sentido más amplio, un script es una serie de comandos que la máquina ejecuta de forma secuencial. En el ámbito de los lenguajes de programación de alto nivel como Python, con frecuencia hace referencia a los archivos que contienen código como scripts.

Secuencia

Una secuencia es un objeto iterable en Python. Se diferencia de una colección en que una secuencia no tiene por qué estar materializada y, por lo tanto, no es accesible directamente. Por ejemplo, la función `range()` retorna una secuencia que puede iterarse, pero sus elementos son inaccesibles por indexación.

Set

Un set o conjunto es una colección de elementos únicos. Informalmente se puede entender como un diccionario formado solo por claves o índices.

Slicing

El *slicing* (o corte) hace referencia a una forma de acceso a colecciones (listas, Series, DataFrame, ndarrays) a través de la cual se obtiene una porción específica de sus elementos.

Streams

Un *stream* hace referencia a un flujo de datos. Se suele emplear el término cuando se trata de lectura y escritura de datos, haciendo referencia al objeto que lleva a cabo dichas operaciones.

Terminal

Un terminal es un programa de ejecución de línea de comandos en el sistema operativo donde el usuario puede interactuar con el sistema a través del teclado. En Windows es cmd (línea de comandos), en MacOS y Linux es la terminal de comandos.

Tipos de elementos

El tipo de una variable determina el tipo de elemento al que hace referencia. En lenguajes de tipo dinámico como Python, una variable puede cambiar de tipo a lo largo de la ejecución del programa. Ejemplos de tipos son `int`, `float`, `string`, `bool`.

Tupla

Una tupla es una colección de elementos inmutables en Python. Toma la forma `(a, b, c)`, donde `a`, `b` y `c` son elementos (pueden ser listas u objetos).

Variable

En programación, una variable es una referencia a un objeto. El contenido de las variables o el elemento al que hacen referencia pueden cambiar. Se puede entender como un cajón en el cual se puede guardar un elemento.



Enlaces de interés

Documentación oficial de Python 3.7

Página oficial en la que se incluye la documentación de todas las versiones de Python. Dispone de material en varios idiomas, entre los que se encuentran el inglés y el francés. Además de documentación sobre la API de Python, se pueden encontrar tutoriales, preguntas frecuentes y novedades.

<https://docs.python.org/3/>

El libro de los antipatrones en Python

Libro de acceso gratuito con una colección de recomendaciones para escribir código en Python. Incluye aspectos estilísticos, de seguridad, rendimiento, etc. El material está en inglés.

<https://docs.quantifiedcode.com/python-anti-patterns/>

Librería Scikit-learn

Módulo en Python para el análisis y la minería de datos, con una gran colección de herramientas útiles para el aprendizaje automático. El enlace incluye documentación y ejemplos de cómo trabajar con la librería en cada uno de los dominios. Hace uso de NumPy y Matplotlib.

<http://scikit-learn.org/stable/>

Documentación oficial de NumPy y SciPy

Página oficial mantenida por la comunidad en la que se encuentra documentación sobre dos de las librerías más utilizadas en computación científica, NumPy y SciPy. El material está en inglés y contiene guías de referencia, manuales y guías para desarrolladores.

<https://docs.scipy.org/doc/>

Documentación oficial de pandas

Página oficial de la librería de análisis de datos por excelencia: pandas. El material está en inglés y proporciona documentación, así como formas de interactuar con la comunidad de desarrolladores y usuarios de pandas, tanto para resolver dudas como para involucrarse en la mejora del módulo.

<https://pandas.pydata.org/index.html>

Documentación oficial de Matplotlib

Página oficial de la librería Matplotlib, la más popular para la representación visual de datos. Está en inglés e incluye ejemplos, tutoriales y guías oficiales para utilizar el módulo de forma adecuada.

<https://matplotlib.org/>

Bibliografía



Abadi, M. et al. (noviembre, 2016). TensorFlow: A System for Large-Scale Machine Learning. En K. Keeton, T. Roscoe (Presidencia), 12th USENIX Symposium on Operating Systems Design and Implementation (pp. 268-283). Savannah, EE. UU.

BBVA Open4U (agosto, 2016). Ventajas e inconvenientes de Python y R para la ciencia de datos. Recuperado de <https://bbvaopen4u.com/es/actualidad/ventajas-e-inconvenientes-de-python-y-r-para-la-ciencia-de-datos>

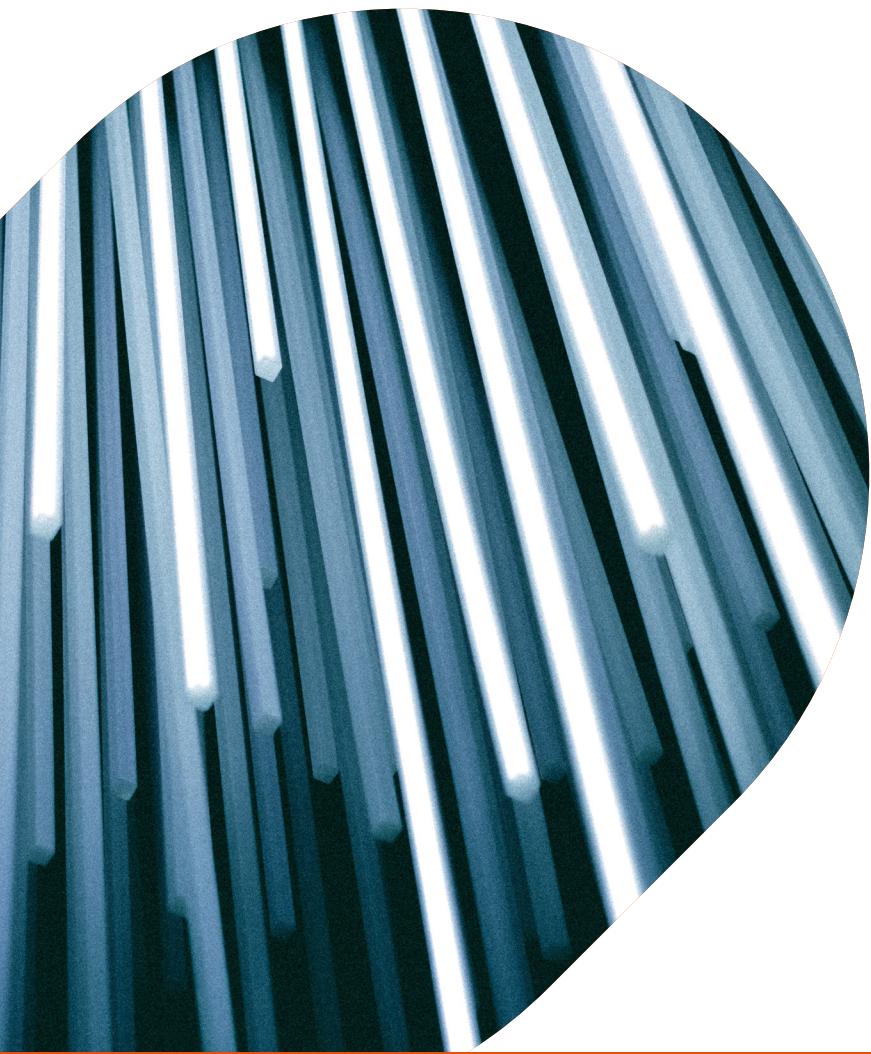
Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I., Bergeron, A. y Bengio, Y. (2011). Theano: Deep Learning on GPUs with Python. *Journal for Machine Learning Research*, 1, 1-48.

Chollet, F. (2015). Keras. Recuperado de <https://keras.io>

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. y Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM international conference on Multimedia*. 675-678.

López, R. (2018). Libro online de IAAR. Recuperado de <https://iaarbook.github.io/>

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V. y Thirion, B. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.



D. Iván Fuertes Torrecilla