BIG DATA CON PYTHON

Recolección, almacenamiento y proceso

Rafael Caballero, Enrique Martín y Adrián Riesco

Alfaomega

libros R

Libro disponible en: eybooks.com

BIG DATA CON PYTHON

Recolección, almacenamiento y proceso

Rafael Caballero Roldán
Enrique Martín Martín
Adrián Riesco Rodríguez

Universidad Complutense de Madrid

Alfaomega

libros R

Diseño de colección y pre-impresión:
Grupo RC
Diseño cubierta:
Cuadratín

Datos catalográficos

Caballero, Rafael; Martín, Enrique; Riesco, Adrián
Big Data con Python. Recolección, almacenamiento y proceso.
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-538-371-2
Formato: 17 x 23 cm Páginas: 284

Big Data con Python. Recolección, almacenamiento y proceso.
Rafal Caballero Roldán, Enrique Martín Martín y Adrián Riesco Rodríguez
ISBN: 978-84-948972-0-7 edición original publicada por RC Libros, Madrid, España.

Primera edición: Alfaomega Grupo Editor, México, diciembre 2018

Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720, Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-538-371-2

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele. d e s c a r g a do en: e y b o oks.c o m

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, C.P. 06720, Del. Cuauhtémoc, Ciudad de México – Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215, piso 10, CP: 1055, Buenos Aires, Argentina, Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

ÍNDICE

PRÓLOGO XI

CAPÍTULO 1. LECTURA DE FICHEROS 1

Introducción 1

CSV 2

TSV 7

Excel 8

JSON 15

XML 19

Conclusiones 24

Referencias 24

CAPÍTULO 2. WEB SCRAPING 25

Introducción 25

Ficheros incluidos en la página web 27

URIs, URLs y URNs 27

Ejemplo: datos de contaminación en Madrid 28

Datos que forman parte de la página 32

Lo que oculta una página web 32

Un poco de HTML 34

Navegación absoluta 38

Navegación relativa 40

Ejemplo: día y hora oficiales 41

Datos que requieren interacción 43

Selenium: instalación y carga de páginas 44

Clic en un enlace 46

Cómo escribir texto 47

Pulsando botones 48

Localizar elementos 50

XPath 51

Navegadores *headless* 58

Conclusiones 58

Referencias 59

CAPÍTULO 3. RECOLECCIÓN MEDIANTE APIS 61

Introducción 61

API Twitter 62

Acceso a Twitter como desarrollador 62

Estructura de un tweet 65

Descargando tweets 69

API-REST 72

Ejemplo: API de Google Maps 72

Ejemplo: API de OMDB 73

Referencias 75

CAPÍTULO 4. MONGODB 77

Introducción 77

¿De verdad necesito una base de datos? ¿Cuál? 78

Consultas complejas 79

Esquema de datos complejo o cambiante 80

Gran volumen de datos 81

Arquitectura cliente-servidor de MongoDB 81

Acceso al servidor 81

Puesta en marcha del servidor 82

Bases de datos, colecciones y documentos 84

Carga de datos 85

Instrucción `insert` 85

Importación desde ficheros CSV o JSON 87

Ejemplo: inserción de tweets aleatorios 88

Consultas simples 89

`find`, `skip`, `limit` y `sort` 89

Estructura general de `find` 93

Proyección en `find` 93

Selección en `find` 94

`find` en Python 99

Agregaciones 100

El pipeline 101

$group 101

$match 103

$project 104

Otras etapas: $unwind, $sample, $out, ... 104

$lookup 106

Ejemplo: usuario más mencionado 107

Vistas 108

Update y remove 109

Update total 109

Update parcial 110

Upsert 112

Remove 113

Referencias 114

CAPÍTULO 5. APRENDIZAJE AUTOMÁTICO CON SCIKIT-LEARN 115

Introducción 115

NumPy 115

pandas (*Python Data Analysis Library*)..... 117

El conjunto de datos sobre los pasajeros del Titanic..... 118

Cargar un DataFrame desde fichero 119

Visualizar y extraer información 120

Transformar DataFrames 124

Salvar a ficheros 125

Aprendizaje automático..... 126

Nomenclatura 127

Tipos de aprendizaje 128

Proceso de aprendizaje y evaluación de modelos..... 129

Etapa de preprocesado 133

Biblioteca scikit-learn 136

Uso de scikit-learn..... 136

Preprocesado 137

Clasificación 140

Regresión 142

Análisis de grupos 144

Otros aspectos de scikit-learn 146

Conclusiones 151

Referencias 152

CAPÍTULO 6. PROCESAMIENTO DISTRIBUIDO CON SPARK..... 153

Introducción..... 153

Conjuntos de datos distribuidos resilientes 157
Creación de RDDs 160
Acciones 162
collect, take y count 163
reduce y aggregate 164
Salvar RDDs en ficheros 167
Transformaciones 169
map y flatMap 169
filter 171
RDDs de parejas 172
Transformaciones combinando dos RDDs 175
Ejemplo de procesamiento de RDD 177
Conclusiones 180
Referencias 180
CAPÍTULO 7. SPARKSQL Y SPARKML 181
SparkSQL 181
Creación de DataFrames 182
Almacenamiento de DataFrames 188
DataFrames y MongoDB 190
Operaciones sobre DataFrames 193
Spark ML 212
Clasificación con SVM 214

Regresión lineal 218

Análisis de grupos con k-means 219

Persistencia de modelos 220

Referencias 221

CAPÍTULO 8. VISUALIZACIÓN DE RESULTADOS 223

Introducción 223

La biblioteca matplotlib 223

Gráficas 230

Gráfica circular 230

Gráfica de caja 233

Gráfica de barras 236

Histograma 241

Conclusiones 244

Referencias 245

APÉNDICE. INSTALACIÓN DEL SOFTWARE 247

Introducción 247

Python y sus bibliotecas 247

Windows 10 248

Linux 250

Mac OS 251

MongoDB 253

Windows 10 253

Linux 256

Mac OS 257

Apache Spark y PySpark 258

Windows 10 258

Linux 259

Mac OS 260

ÍNDICE ANALÍTICO 261

PRÓLOGO

Hablar de la importancia del análisis de datos resulta, hoy en día, innecesario. El tema ocupa páginas en los periódicos, las universidades abren nuevos grados y másteres dedicados a la ciencia de datos, y la profesión de analista de datos se encuentra entre las más demandadas por las empresas, que temen quedarse atrás en esta nueva fiebre del oro.

Todos tenemos la sensación de que una multitud de datos se encuentra, ahí, al alcance de la mano, esperando la llegada del experto que sea capaz de transformarlos en valiosa información. Es emocionante, pero a la vez un tanto frustrante, porque no siempre conocemos las palabras mágicas capaces de llevar a cabo el sortilegio.

Y es que los grandes éxitos del análisis de datos llevan por detrás muchas horas de trabajo minucioso, que a menudo no son visibles. Por ejemplo, podemos encontrar una gran cantidad de blogs en la web que nos muestran los excelentes resultados que se obtienen al aplicar tal o cual técnica a cierto conjunto de datos, pero muchos menos que nos hablen del esfuerzo dedicado a obtener, preparar y preprocesar estos datos, tareas que, como todo el mundo que trabaja en esta área ha experimentado, consumen la amplia mayoría del tiempo.

Este libro pretende hablar del análisis de datos examinando la "vida" de los datos paso a paso. Desde su origen, ya sean ficheros de texto, Excel, páginas web, o redes sociales, a su preprocesamiento, almacenamiento en una base de datos, análisis y visualización. Pretendemos mostrar el análisis de datos tal y como es: un área fascinante, pero también una labor que requiere muchas horas de trabajo cuidadoso.

Buscamos además que nuestro texto sea útil en entornos Big Data. Para ello emplearemos bases de datos con escalabilidad horizontal, es decir, con posibilidad de crecer de forma prácticamente ilimitada sin ver afectada su eficiencia, como MongoDB, y entornos de procesamiento capaces de tratar estos datos, como Spark. En todo caso, somos conscientes de que el tema es muy extenso, y que en un solo

libro no cabe todo. Por ello, cada capítulo incluye un apartado de bibliografía con lecturas recomendadas para conocer el tema más a fondo.

El nexo de unión entre todas las etapas, desde el 'nacimiento' del dato hasta su (gloriosa) transformación en información, será el lenguaje Python, el más utilizado dentro del análisis de datos debido a la multitud de bibliotecas que pone a nuestra disposición. Aunque el libro no asume ningún conocimiento previo de análisis de datos, sí supone unos mínimos conocimientos de Python, o al menos de algún lenguaje de programación que nos ayude a comprender el código.

Todos los capítulos de este libro contienen ejemplos detallados de cómo realizar las distintas tareas en Python. Por comodidad para el lector, además de los fragmentos incluidos en el libro también hemos creado *notebooks* de Jupyter para cada capítulo donde incluimos el código completo junto con ejemplos y comentarios adicionales. Estos notebooks, junto con los conjuntos de dato utilizados a lo largo del libro, se pueden encontrar en el repositorio

`https://github.com/RafaelCaballero/BigDataPython`

Además del lenguaje de programación Python, nuestro viaje nos requerirá la utilización de diversas tecnologías adicionales. Aunque en todos los casos son de fácil instalación, hemos añadido un apéndice con instrucciones básicas que esperamos sean de utilidad.

Si ya estamos situados, podemos empezar a estudiar la vida de los datos, comenzando por su nacimiento.

LOS AUTORES

Rafael Caballero es doctor en Ciencias Matemáticas y actualmente dirige la Cátedra de Big Data y Analítica Hewlett Packard-UCM. Profesor de la Facultad de Informática de la Universidad Complutense de Madrid con 20 años de experiencia en docencia de bases de datos y gestión de la información, también es autor de más de 50 publicaciones científicas y de varios libros sobre lenguajes de programación. Aplica su interés por Big Data a los grandes catálogos astronómicos, habiendo descubierto mediante el análisis de estos catálogos más de 500 estrellas dobles nuevas.

Enrique Martín Martín es doctor en Ingeniería Informática por la Universidad Complutense de Madrid, universidad en la que ha sido profesor desde 2007. Durante años ha impartido asignaturas en la Facultad de Informática sobre gestión de la

información y Big Data. Su investigación principal gira en torno a los métodos formales para el análisis de programas en entornos distribuidos.

Adrián Riesco es doctor en Ingeniería Informática por la Universidad Complutense de Madrid, universidad en la que ha sido profesor desde 2011. Su docencia incluye una asignatura de introducción a la programación Python, así como otras asignaturas de grado y máster. Sus principales áreas de investigación son la depuración de programas y los métodos formales basados en lógica de reescritura.

1 LECTURA DE FICHEROS

INTRODUCCIÓN

Para lograr analizar datos y convertirlos en información, el primer paso es ser capaz de incorporarlos a nuestro programa, esto es, *cargar* los datos. En este capítulo discutimos la adquisición de datos desde fichero, por lo que en primer lugar es necesario plantearse una serie de preguntas: ¿qué son *datos*? ¿Su adquisición se limita a descargar datos de internet? ¿Es capaz el lenguaje Python de entender cualquier fuente de información, tales como texto, imágenes, audio y vídeo? ¿Puedo obtener información de cualquier fuente, como páginas oficiales del gobierno, periódicos, redes sociales y foros de opinión?

Aunque en general entendemos por datos cualquier tipo de información que se almacena en un ordenador, en el contexto de este libro usaremos datos para referirnos a colecciones de elementos con una serie de atributos. Así, nuestros datos se pueden referir a un conjunto de personas con DNI, nombre, edad y estado civil, a multas con el identificador del infractor, la matrícula del vehículo, la cuantía a pagar y su estado (pagada o no), o a los productos de un supermercado, con fecha de caducidad, precio, cantidad y ofertas, entre muchos otros. ¿Significa esto que un texto o una imagen no son datos? Sí que lo son, pero en este caso son información *en bruto*: para poder trabajar con ellos necesitaremos una fase de análisis previa que extraiga la información que deseamos, como por ejemplo la frecuencia de aparición de las palabras en el texto o el color y la cantidad de los objetos en la imagen. Esta fase previa, llamada *preprocesado*, puede hacerse de manera eficiente con alguna de las técnicas que veremos en capítulos posteriores, pero por el momento consideraremos que los datos que tenemos no necesitan esta fase.

Pero ¿cómo obtenemos esta información? Es muy posible que ya tengamos en casa información de este tipo, cuando hemos guardado información sobre los jugadores del equipo del barrio, sobre nuestros gastos fijos para ver cómo ahorrar o sobre los temas a evitar en las cenas de Navidad. No vale cualquier forma de guardar esta información, claro, para poder trabajar con los datos es necesario que estos estén *estructurados* o al menos *semi-estructurados*. Si para nuestra la liga del barrio guardamos los goles y los minutos jugados por Juanito y Pepito y lo hacemos de esta forma:

```
Juanito este año ha jugado 200 minutos, marcando 3 goles.
Por su parte, Pepito solo ha jugado 100' pero ha marcado 10 goles.
```

La falta de estructura hará que procesar la información sea muy difícil. Sería mucho más sencillo tener la información guardada, por ejemplo, de la siguiente manera:

```
Juanito, 200, 3
Pepito, 100, 10
```

Esta manera de guardar la información tiene una estructura fija, por lo que es fácil extraer los atributos de cada elemento. Generalizando esta idea, y con el objetivo de facilitar la compartición de la información y automatizar su uso, a lo largo de los años han ido surgiendo distintos estándares para el almacenamiento. En este capítulo veremos cómo cargar en Python ficheros almacenados siguiendo los formatos más populares: CSV, TSV, MS Excel, JSON y XML. Los ficheros que usaremos para ilustrar este proceso contienen información pública ofrecida por el Gobierno de España en `http://datos.gob.es/es`. En particular, usaremos la información sobre las subvenciones asignadas en 2016 a asociaciones del ámbito educativo por el Ayuntamiento de Alcobendas. En los capítulos siguientes veremos cómo descargar información en estos formatos desde distintas fuentes, como redes sociales, páginas web o foros.

CSV

El formato CSV, del inglés *Comma Separated Values* (Valores Separados por Comas) requiere que cada elemento de nuestro conjunto se presente en una línea. Dentro de esa línea, cada uno de los atributos del elemento debe estar separado por un único separador, que habitualmente es una coma, y seguir siempre el mismo orden. Además, la primera línea del fichero, a la que llamaremos *cabecera*, no contiene datos de ningún elemento, sino información de los atributos. Por ejemplo, un profesor de instituto podría guardar las notas de sus alumnado en formato CSV como sigue:

```
Nombre, Lengua, Física, Química, Gimnasia
Alicia Alonso, Notable, Notable, Aprobado, Bien
Bruno Bermejo, Aprobado, Bien, Bien, Suspenso
```

En este caso es fácil separar los campos, pero ¿qué ocurre si queremos guardar la nota numérica y esta es 7,5? El formato del fichero no es capaz de distinguir entre la coma que marca el inicio de la parte decimal de la nota y la coma que separa los distintos atributos, y podría pensar que hay dos notas: 7 y 5. Podemos solucionar este problema de dos maneras. En primer lugar, es posible crear un valor único encerrándolo entre comillas, por lo que tendríamos:

```
Nombre, Lengua, Física, Química, Gimnasia
"Alicia Alonso", "7,5", "8", "5,5", "6,5"
"Bruno Bermejo", "5,5", "6,5", "6,25", "4,75"
```

En este caso, entendemos que el atributo es un único valor de tipo cadena de texto (`str`) y por tanto no debe partirse. Esta solución es tan general que es habitual introducir todos los valores entre comillas, incluso cuando tenemos la seguridad de que no contienen ninguna coma. En particular, es una práctica segura cuando estamos generando ficheros en este formato de manera automática. La segunda opción consiste en cambiar la coma, separando los valores por otro símbolo que tengamos la seguridad no aparece en el resto del fichero, como un punto o una arroba. Esta opción es más arriesgada si no conocemos bien el fichero, por lo que es recomendable usar las comillas.

Veamos ahora cómo cargar en Python un fichero con formato CSV. Usaremos, como indicamos arriba, la información sobre subvenciones en el ámbito educativo asignadas en 2016 por el Ayuntamiento de Alcobendas. El fichero tiene la información como sigue:

```
"Asociación", "Actividad Subvencionada ", "Importe"
"AMPA ANTONIO MACHADO","TALLER FIESTA DE CARNAVAL", "94.56"
"AMPA ANTONIO MACHADO", "TALLER DIA DEL PADRE","39.04"
...
```

Como se puede observar, la primera columna indica el nombre de la asociación, la segunda el nombre de la actividad y la tercera el importe asignado. Es interesante observar que todos los valores están encerrados en comillas para evitar errores y que el nombre de la segunda columna contiene un espacio antes de cerrar las comillas, por lo que este espacio será parte del nombre, y puede suponer problemas si intentamos acceder a esa posición escribiendo el nombre de la columna manualmente. Para manipular este tipo de ficheros usaremos la biblioteca `csv` de

Python, por lo que para el resto del capítulo consideraremos que la hemos cargado mediante:

```
>>> import csv
```

La manera más sencilla de cargar un fichero de este tipo es abrirlo con `open` y crear un lector con `reader`. El lector obtenido es iterable y cada elemento se corresponde con una línea del fichero cargado; esta línea está representada como una lista donde cada elemento es una cadena de caracteres que corresponde a una columna. Así, si queremos, por ejemplo, calcular el importe total que se dedicó a subvenciones deberemos usar un programa como el siguiente, donde `ruta` es la dirección al fichero correspondiente:[1]

```
>>> with open(ruta, encoding='latin1') as fichero_csv:
>>>     lector = csv.reader(fichero_csv)
>>>     next(lector, None)  # Se salta la cabecera
>>>     importe_total = 0
>>>     for linea in lector:
>>>         importe_str = linea[2]
>>>         importe = float(importe_str)
>>>         importe_total = importe_total + importe
>>>     print(importe_total)
```

En este código podemos resaltar los siguientes elementos:

- Hemos abierto el fichero fijando el `encoding` a `latin1`. Esta codificación nos permite trabajar con tildes, por lo que su uso es interesante en documentos en español.
- Es necesario transformar de cadena a número usando `float`, ya que el lector carga todos los valores como cadenas.
- El lector incluye la fila con la cabecera, que es necesario saltarse para no producir un error cuando tratamos de transformar en número la cadena `'Importe'`.

El problema de esta representación radica en la necesidad de usar índices para acceder a los valores, con lo que el código resultante es poco intuitivo. Para mejorar este aspecto podemos usar `DictReader` en lugar de `reader`, la cual devuelve un lector en el que cada línea es un diccionario con las claves indicadas en la cabecera del documento y valor el dado en la fila correspondiente. De esta manera, sería

[1] Para facilitar la lectura del código, no escribiremos en general las rutas a los ficheros en el capítulo. Tanto los ficheros de prueba como los resultados están disponibles en la carpeta `Cap1` del repositorio GitHub.

posible sustituir la expresión `linea[2]` que hemos visto en el código anterior por `linea['Importe']`. Vamos a usar esta nueva manera de leer ficheros para calcular un diccionario que nos indique, para cada asociación, el importe total de subvenciones asignado:

```
>>> with open(ruta, encoding='latin1') as fichero_csv:
>>>     dict_lector = csv.DictReader(fichero_csv)
>>>     asocs = {}
>>>     for linea in dict_lector:
>>>         centro = linea['Asociación']
>>>         subvencion = float(linea['Importe'])
>>>         if centro in asocs:
>>>             asocs[centro] = asocs[centro] + subvencion
>>>         else:
>>>             asocs[centro] = subvencion
>>>     print(asocs)
```

En esta ocasión no ha sido necesario saltarse la primera línea, ya que `DictReader` entiende que dicha línea contiene los nombres de los campos y es usado para crear las claves que serán posteriormente usadas en el resto de líneas. Puede parecer que este comportamiento limita el uso de `DictReader`, ya que en principio nos impide trabajar con ficheros sin cabecera, pero esta limitación se supera usando `fieldnames=cabecera` en `DictReader` (también disponible en `reader`), donde `cabecera` es una lista de cadenas que indica los nombres y el orden de los valores en el fichero, y cuya longitud se debe corresponder con el número de columnas. En el caso concreto del código anterior, si nuestro fichero no tuviese cabecera, modificaríamos la llamada a `DictReader` y escribiríamos:

```
>>> csv.DictReader(fichero_csv,fieldnames=['Asociación',
              'Actividad Subvencionada ', 'Importe']).
```

Una vez el fichero ha sido cargado, estamos interesados en manipularlo y almacenar los resultados obtenidos. Igual que al leer un fichero, podemos escribir en dos modalidades: con objetos escritores devueltos con la función `writer()` o con instancias de la clase `DictWriter`. En ambos casos podemos usar los argumentos que usamos en los lectores, como `fieldnames`, y disponemos de las funciones `writerow(fila)` y `writerows(filas)`, donde `filas` hace referencia a una lista de `fila`, que se define de distinta manera según el caso:

- Cuando tratamos con objetos generados por `writer()` una `fila` es un iterable de cadenas de texto y números.
- Cuando tratamos con objetos de la clase `DictWriter` una `fila` es un diccionario cuyos valores son cadenas de texto y números.

Además, los objetos de la clase `DictWriter` ofrecen otra función `writeheader()`, que escribe en el fichero la cabecera de la función, previamente introducida con el parámetro `fieldnames`.

Veamos un ejemplo de cómo modificar y guardar un nuevo fichero. En particular, vamos a añadir a nuestro fichero de subvenciones dos nuevas columnas, `Justificación requerida` y `Justificación recibida`. En la primera almacenaremos `Sí` si la subvención pasa de 300 euros y `No` en caso contrario; en la segunda pondremos siempre `No`, ya que todavía no hemos recibido justificación alguna. Empezamos abriendo los ficheros y creando un objeto lector como en los ejemplos anteriores para recorrer el fichero original:

```
>>> ruta1 = 'src/data/Cap1/subvenciones.csv'
>>> ruta2 = 'src/data/Cap1/subvenciones_esc.csv'
>>> with open(ruta1, encoding='latin1') as fich_lect,     open(ruta2,
'w', encoding='latin1') as fich_escr:
>>>     dict_lector = csv.DictReader(fich_lect)
```

A continuación, extraemos los nombres en la cabecera del lector, valor obtenido como una lista, y le añadimos las dos columnas que deseamos añadir. Con esta nueva lista creamos un objeto escritor; dado que estamos leyendo la información como un diccionario lo más adecuado es crear un escritor de la clase `DictWriter`:

```
>>>     campos = dict_lector.fieldnames +
   ['Justificación requerida', 'Justificación recibida']
>>>     escritor = csv.DictWriter(fich_escr, fieldnames=campos)
```

Para almacenar la información, empezaremos escribiendo la cabecera en el fichero, para después continuar con un bucle en el que extraemos la línea actual del lector, comprobando si el importe es mayor de 300 euros, en cuyo caso añadimos el campo correspondiente al diccionario con el valor `Sí`; en otro caso, lo añadiremos con el valor `No`. De la misma manera, añadimos siempre el campo sobre justificación con el valor `No`:

```
>>>     escritor.writeheader()
>>>     for linea in dict_lector:
>>>        if float(linea['Importe']) > 300:
>>>           linea['Justificación requerida'] = "Sí"
>>>        else:
>>>           linea['Justificación requerida'] = "No"
>>>        linea['Justificación recibida'] = "No"
```

Por último, usamos la función `writerow()` para volcar el diccionario en el fichero. Otra posible opción sería almacenar cada línea en una lista y acceder una única vez a fichero usando la función `writerows()`.

```
>>>         escritor.writerow(linea)
```

Si abrimos el fichero y observamos el resultado podemos ver que el fichero se ha creado correctamente. Además, también observamos que los objetos escritores comprueban cuándo un campo contiene una coma y, en dichos casos, encierra los valores entre comillas para asegurarse que sea leído posteriormente, por lo que el usuario no necesita preocuparse por estos detalles:

```
Asociación, Actividad Subvencionada, Importe, Justificación
requerida, Justificación recibida
...
AMPA CASTILLA,"PROPUESTA DE ACTIV EXTRAESCOLARES, INSCRIPCIONES,
INTERCAMBIO LIBROS",150,No,No
...
```

Con este ejemplo cubrimos los aspectos básicos de la biblioteca `csv`. Sin embargo, indicamos al principio de la sección que el separador de columnas en este tipo de ficheros no es obligatoriamente una coma; en la próxima sección veremos una variante de este formato y cómo manejarlo con la misma biblioteca `csv`.

TSV

El formato TSV, del inglés *Tab Separated Values* (Valores Separados por Tabuladores), es una variante de CSV donde se sustituyen las comas separando columnas por tabuladores. Aunque este formato es menos común que CSV, es interesante tratarlo porque nos muestra cómo usar la biblioteca `csv` para manejar otro tipo de ficheros. En efecto, en la página del Gobierno de España de la que estamos extrayendo los datos para este capítulo solo encontramos dos ficheros TSV (de hecho, uno de ellos está vacío, por lo que en la práctica solo hay uno) y ninguno cubre todos los formatos que queremos tratar, por lo que crearemos nuestro propio fichero TSV a partir del fichero CSV utilizado en la sección anterior. Para ello vamos a hacer uso de otro de los argumentos de los objetos lectores y escritores, `delimiter`. Este argumento, que por defecto está asignado a una coma, indica cuál es el valor que se usará para separar columnas. Si creamos un objeto escritor con el argumento `delimiter='\t'` dicho objeto escribirá ficheros TSV. Usamos esta idea en el ejemplo siguiente, en el cual leemos un fichero CSV (y por tanto no modificamos `delimiter`) pero escribimos un fichero TSV:


```
>>> with open(ruta1, encoding='latin1') as fich_lect,
     open(ruta2, 'w', encoding='latin1') as fich_escr:
>>>    dict_lector = csv.DictReader(fich_lect)
>>>    campos = dict_lector.fieldnames
>>>    escritor = csv.DictWriter(fich_escr, delimiter='\t',
>>>       fieldnames=campos)
>>>    escritor.writeheader()
>>>    for linea in dict_lector:
>>>       escritor.writerow(linea)
```

Una vez usado este argumento, el resto del código es igual al utilizado para CSV. Por ejemplo, el código siguiente calcula el mismo diccionario relacionando asociaciones y subvención total que mostramos en la sección anterior:

```
>>> with open(ruta2, encoding='latin1') as fich:
>>>    dict_lector = csv.DictReader(fich, delimiter='\t')
>>>    asocs = {}
>>>    for linea in dict_lector:
>>>       centro = linea['Asociación']
>>>       subvencion = float(linea['Importe'])
>>>       if centro in asocs:
>>>          asocs[centro] = asocs[centro] + subvencion
>>>       else:
>>>          asocs[centro] = subvencion
>>>    print(asocs)
```

EXCEL

El formato XLS, utilizado por Microsoft Excel, ofrece una estructura de tabla algo más flexible que los formatos anteriores e incorpora la idea de *fórmulas*, que permiten calcular valores en función de otros en la misma tabla. Python no tiene una biblioteca estándar para manipular este tipo de ficheros, por lo que en esta sección presentaremos `xlrd` y `xlwt`, los estándares *de facto* para estos ficheros. Al final de la sección también describiremos brevemente cómo utilizar la biblioteca `pandas`, que usaremos en capítulos posteriores, para leer y escribir ficheros XLS.

Recordemos para empezar cómo se organiza un fichero Excel. La estructura principal es un *libro*, que se compone de diversas *hojas*. Cada una de estas hojas es una matriz de celdas que pueden contener diversos valores, en general cadenas de caracteres, números, Booleanos, fórmulas y fechas.

A la hora de leer ficheros usando la biblioteca `xlrd` (de *XLs ReaD*) cada uno de estos elementos tiene su correspondencia en Python como sigue:

- La clase `Book` representa libros. Los objetos de esta clase contienen información sobre las hojas del libro, la codificación, etc. En la biblioteca `xlrd` no podemos crear objetos de esta clase, deberemos obtenerlos leyendo desde un fichero. Sus atributos y métodos principales s on:
  - La función `open_workbook(ruta)`, que devuelve el libro correspondiente al fichero almacenado en `ruta`.
  - El atributo `nsheets`, que devuelve el número de hojas del libro. Podemos usar este valor para acceder a las hojas en un bucle usando la función a continuación.
  - La función `sheet_by_index(indx)`, que devuelve un objeto de clase `Sheet` correspondiente a la hoja identificada por el índice `indx` dado como argumento.
  - La función `sheet_names()`, que devuelve una lista con los nombres de las hojas en el libro. Podemos usar esta lista para acceder a las hojas usando la función a continuación.
  - La función `sheet_by_name(nombre)`, que devuelve un objeto de clase `Sheet` que se corresponde a la hoja que tiene por nombre el valor dado como argumento.
  - La función `sheets()`, que devuelve una lista de objetos de clase `Sheet` con todas las hojas del libro.
- La clase `Sheet` representa hojas. Los objetos de esta clase contienen información sobre las celdas de la hoja y, como ocurría en el caso anterior, no pueden ser creados directamente, sino que son obtenidos a partir de libros. Los principales atributos y métodos de esta clase son:
  - Los atributos `ncols` y `nrows`, que indican el número de columnas y filas en la hoja, respectivamente.
  - El atributo `name`, que indica el nombre de la hoja.
  - La función `cell(filx, colx)`, que devuelve la celda en la posición dada por `filx` y `colx`. Es importante observar aquí varias diferencias entre el uso habitual de tablas Excel y los índices usados en Python. Cuando trabajamos con una tabla Excel la primera fila tiene índice 1, en Python es la fila con índice 0. Asimismo, la primera columna en Excel es A, mientras en Python está identificada por 0. Deberem os tener muy en cuenta estas diferencias cuando creemos fórmulas directamente desde Python.
  - La función `col(colx)` devuelve un iterable con todas las celdas contenidas en la columna dada como argumento. De la misma manera, la función `row(rowx)` devuelve todas las celdas en la fila indicada.
- La clase `Cell` representa celdas. Esta clase contiene información sobre el tipo, el valor y el formato de la información que contiene cada celda en

una hoja. Como en los casos anteriores, no se crean objetos de esta clase directamente, sino que se extraen de las hojas con las funciones vistas arriba. Los atributos de esta clase son:

- El atributo `ctype` indica el tipo de la celda. Este valor es un número entero que se interpreta como sigue:

0	Celda vacía
1	Cadena de caracteres
2	Número
3	Fecha
4	Booleano, donde 1 indica `True` y 0 indica `False`.
5	Error interno de Excel
6	Casilla con valor vacío (p.e., la cadena vacía).

- El atributo `value` indica el valor contenido en la celda.
- El atributo `xf_index` indica el formato por defecto para las celdas. Solo cuando está definido podemos tener celdas de tipo 6.

– Además, la biblioteca proporciona las funciones `colname(col)` y `cellname(fil, col)`, que devuelven el nombre de la columna identificada por `col` y de la celda identificada por `fil` y `col`, respectivamente. Así, el resultado de `colname(2)` es `C` y de `colname(35)` es `AJ`, mientras que `cellname(3, 2)` es `C4` es y `cellname(7, 35)` es `AJ8`.

Por otra parte, usaremos la biblioteca `xlwt` (*XLs WriTe*) para escribir ficheros. Esta biblioteca ofrece funciones sobre libros, hojas y ficheros:

– La clase `Workbook` representará los nuevos libros. Las funciones principales de la clase son:
  - La función `xlwt.Workbook()`, que crea un nuevo `Workbook`.
  - La función `libro.add_sheet(nom)`, que crea una nueva hoja y la añade al libro. La función devuelve un objeto `WorkSheet` que corresponde a la hoja recién creada.
  - La función `libro.save(ruta)`, que guarda en disco, en la dirección `ruta`, el `libro` usado para llamar a la función.

– La clase `WorkSheet` representa, como se ha indicado arriba, las hojas. Su principal función es la función `hoja.write(fil, col, val)`, que escribe el valor `val` en la celda situada en la fila `fil` y columna `col`.

– Finalmente, la clase `Formula` nos permite definir nuevas fórmulas. Para ello usaremos el constructor `Formula(form)`, donde `form` es una cadena con la fórmula que queremos escribir. Por ejemplo, `Formula("A1*V3")` crea la fórmula que multiplica los valores en las casillas A1 y V3.

Es importante tener claro que los libros y las hojas de las distintas bibliotecas son diferentes y no compatibles, deberemos usar unos u otras dependiendo de si queremos leer o escribir el fichero.

Con esta información estamos listos para trabajar con ficheros Excel. Durante el resto de la sección supondremos que tenemos cargadas las bibliotecas correspondientes como:

```
import xlrd
import xlwt
```

Como en capítulos anteriores, usaremos la información sobre subvenciones a asociaciones del ámbito educativo. En esta ocasión, el fichero utilizado contiene la información en una sola hoja, llamada `Hoja 1`, de la forma

	A	B	C
1	Asociación	Actividad Subvencionada	Importe
2	AMPA ANTONIO MACHADO	TALLER FIESTA DE CARNAVAL	94,56
3	AMPA ANTONIO MACHADO	TALLER DIA DEL PADRE	39,04

Obsérvese que, por ejemplo, la casilla A3 (con valor `AMPA ANTONIO MACHADO`) estará identificada por los índices 2 (fila) y 0 (columna) y tendrá tipo 1.

Empezaremos usando la biblioteca `xlrd`. Para ello implementaremos, como ya hicimos en CSV y TSV, la funcionalidad necesaria para crear un diccionario que tenga como claves las distintas asociaciones y como valores el importe total de subvenciones obtenidas. En primer lugar, obtenemos el libro con la función `open_workbook` e inicializamos el diccionario que queremos calcular:

```
>>> with xlrd.open_workbook(ruta) as libro:
>>>     asocs = {}
```

A continuación, recorreremos las hojas del libro. Como este libro solo contiene una hoja podríamos acceder a ella directamente usando las funciones `libro.sheet_by_index(0)` o `libro.sheet_by_name("Hoja 1")`, pero usamos un bucle `for` para ilustrar cómo funcionaría un caso más general, donde tenemos varias hojas con datos, quizás de distintos ayuntamientos:

```
>>>     for hoja in libro.sheets():
```

Una vez en la hoja procedemos a recorrer las filas. Para ello, usamos un bucle `for` sobre el número de filas, teniendo cuidado de empezar en 1 para saltarnos los nombres de las columnas. La primera instrucción dentro del bucle será obtener la fila correspondiente al índice con la función `hoja.row(i)`:

```
>>>     for i in range(1,hoja.nrows):
>>>         fila = hoja.row(i)
```

Solo nos queda extraer los valores de la fila y actualizar el diccionario. Obsérvese que en este caso la celda correspondiente al informe "sabe" que contiene un valor numérico (es decir, su tipo es 2) y no es necesario obtener el valor correspondiente a partir de una cadena:

```
>>>         asoc = fila[0].value
>>>         subvencion = fila[2].value
>>>         if centro in asocs:
>>>             asocs[asoc] = asocs[asoc] + subvencion
>>>         else:
>>>             asocs[centro] = subvencion
```

Para aprovechar la potencia de Excel e ilustrar el uso de fórmulas vamos a crear, a partir del libro que tenemos, un nuevo libro con dos hojas. En la primera copiaremos exactamente el contenido de la hoja actual, mientras que la segunda contendrá una tabla con cuatro columnas. La primera será el nombre de la asociación, la segunda el total del importe recibido en subvenciones, la tercera el total justificado (inicialmente 0 para todas) y la cuarta será el importe que queda por justificar, que consistirá en una fórmula que resta lo justificado hasta el momento (tercera columna) menos el total (segunda columna).

Empezamos el código cargando, como hicimos anteriormente, en `libro_lect` el libro e inicializando el diccionario de asociaciones:

```
>>> with xlrd.open_workbook(ruta) as libro_lect:
>>>     asocs = {}
```

A continuación creamos un nuevo libro con el constructor `Workbook()`:

```
>>>     libro_escr = xlwt.Workbook()
```

El libro lo rellenaremos usando los mismos nombres para las hojas que el libro original, por lo que en esta ocasión usaremos un bucle `for` recorriendo los nombres de las hojas. Para cada nombre, extraemos la hoja correspondiente del libro original y creamos otra con el mismo nombre en el libro destino con `add_sheet(nombre)`:

```
>>>     for nombre in libro_lect.sheet_names():
>>>        hoja_lect = libro_lect.sheet_by_name(nombre)
>>>        hoja_escr = libro_escr.add_sheet(nombre)
```

Para cada libro recorremos ahora todas las celdas con dos bucles anidados, accediendo a cada celda de la fuente y escribiendo su valor en el destino.

```
>>>       for i in range(hoja_lect.nrows):
>>>          for j in range(hoja_lect.ncols):
>>>             valor = hoja_lect.cell(i,j).value
>>>             hoja_escr.write(i, j, valor)
```

Además, para cada fila que no sea la inicial (que contiene la cabecera, pero no valores), vamos a calcular, como hemos hecho anteriormente, el diccionario que indica, para cada asociación, la subvención total. Usaremos este diccionario para rellenar la segunda hoja:

```
>>>         if i != 0:
>>>             fila = hoja_lect.row(i)
>>>             centro = fila[0].value
>>>             sub = fila[2].value
>>>             if centro in asocs:
>>>                 asocs[centro] = asocs[centro] + sub
>>>             else:
>>>                 asociaciones[centro] = sub
```

Una vez tenemos el diccionario con los costes totales pasamos a escribir la segunda hoja, que llamaremos `Totales`. En la primera fila escribimos los valores de las columnas:

```
>>>    hoja_escr = libro_escr.add_sheet('Totales')
>>>    hoja_escr.write(0, 0, "Asociación")
>>>    hoja_escr.write(0, 1, "Importe total")
>>>    hoja_escr.write(0, 2, "Importe justificado")
>>>    hoja_escr.write(0, 3, "Restante")
```

Para rellenar el resto de la hoja recorremos, con índice, el diccionario. Como ya hemos escrito la cabecera deberemos sumar 1 al índice para escribir en la fila correcta. Los valores en las primeras dos columnas los extraemos del diccionario, mientras que el tercero siempre es 0, por lo que podemos escribirlos directamente:

```
>>>    for i, clave in enumerate(asocs):
>>>        fila = i + 1
>>>        hoja_escr.write(fila, 0, clave)
>>>        hoja_escr.write(fila, 1, asocs[clave])
>>>        hoja_escr.write(fila, 2, 0)
```

El cuarto valor, sin embargo, es una fórmula que depende de la fila en la que estamos, por lo que debemos elaborarlo algo más. Como indicamos arriba, la representación estándar de Excel y la representación de Python para los índices de las filas difieren por uno, por lo que deberemos sumar 2 al índice (1 porque ya hemos

escrito la cabecera y 1 más por la diferencia en las representaciones) para elegir la fila correcta. Después crearemos una cadena de texto con la fórmula deseada, que para la fila `i` será `Ci-Bi`, y usamos la cadena resultante como argumento para el constructor de fórmulas, que será el valor que introduciremos en la celda.

```
>>>         fila_form = i + 2
>>>         fform_str = str(fila_form)
>>>         form = "C" + fform_str + "-B" + fform_str
>>>         hoja_escr.write(fila, 3, xlwt.Formula(form))
```

Por último, usamos la función `save` para guardar el resultado en disco:

```
>>>     libro_escr.save(ruta2)
```

La hoja resultante tiene la siguiente estructura, donde el valor en `Restante` se modificará automáticamente cuando varíe el importe justificado:

	A	B	C	D
1	Asociación	Importe total	Importe justificado	Restante
2	AMPA ANTONIO MACHADO	2344,99	0	-2344,99
3	AMPA BACHILLER ALONSO LOPEZ	3200	0	-3200

Como indicamos arriba, hemos podido crear la fórmula directamente porque los nombres de las columnas eran fijas. Si este no es el caso, recordemos que podemos usar las funciones `colname(col)` y `cellname(fil, col)` de la biblioteca `xlrd` para obtener el nombre de la columna.

Por último, exponemos brevemente cómo usar la biblioteca `pandas` para cargar y almacenar *dataframes*. Los libros se cargan con la función `ExcelFile(ruta)`, que devuelve un objeto que proporciona información sobre las hojas del libro (atributo `sheet_names`) y que puede cargar una hoja concreta como un *dataframe* con la función `parse(nombre)`. Para grabar podemos crear un objeto escritor con la función `ExcelWriter(ruta)` y usarlo como argumento de la función de *dataframes* `to_excel(escritor, nombre_hoja)` para añadir hojas. Por último, la función `save` almacenará en disco el libro resultante. Como ejemplo, el código siguiente realiza una copia de un libro dado:

```
>>> import pandas
>>> with pandas.ExcelFile(ruta1) as xl:
>>>    escritor = pandas.ExcelWriter(ruta2)
>>>    for nombre in xl.sheet_names:
>>>      df = xl.parse(nombre)
>>>      df.to_excel(escritor,nombre)
>>>   escritor.save()
```

JSON

A diferencia de los formatos anteriores, que compartían una estructura fija de tabla, los formatos restantes (JSON y XML) son más flexibles y nos permitirán almacenar la información semiestructurada. JSON, acrónimo de *JavaScript Object Notation* (notación de objetos de JavaScript), permite que los datos se almacenen de dos maneras:

- Como *objetos*, que consisten en parejas `nombre : valor` separadas por comas y encerradas entre llaves. Por ejemplo, un objeto con los datos de una persona (nombre, apellidos, edad) podría tener la forma `{"nombre":"Fulanito", "apellido":"García", "edad": 10}`. Como se observa en el ejemplo, el primer elemento de las parejas es siempre una cadena, mientras que el segundo es un *valor*, que definimos más abajo.
- Como *arrays* de valores, separados por comas y encerrados entre corchetes. Por ejemplo, una lista de números `[3, 0, 10, 3.4]`.
- Es importante observar que consideramos las parejas en los objetos como un *conjunto*, esto es, el orden de las parejas no es importante para los datos, mientras que los valores en los *arrays* sí están ordenados.

Los valores válidos en un documento JSON son cadenas, números, Booleanos (`true` y `false`), la constante `null`, objetos y *arrays*. Es fácil ver que estos elementos tienen su análogo en Python: las cadenas, números y Booleanos se traducen directamente (cambiando `true` por `True` y `false` por `False`), `null` pasa a ser `None`, los objetos a diccionarios y los *arrays* serán listas.

Usando objetos y *arrays* como valores dentro de otros objetos/*arrays* podemos crear estructuras complejas. Sin embargo, el fichero de subvenciones con el que trabajamos no hace uso de estas posibilidades y está formado por una lista de objetos, cada uno de los cuales se corresponde con una fila de los formatos anteriores, por lo que la información sobre la asociación se encuentra repetida y las actividades correspondientes se hallan repartidas entre varios objetos cuando podrían estar dentro de uno solo:

```
[
  {
    "Asociación":"AMPA ANTONIO MACHADO",
    "Actividad Subvencionada":"TALLER FIESTA DE CARNAVAL",
    "Importe en euros":"94.56"
  },
  {
```

```
    "Asociación":"AMPA ANTONIO MACHADO",
    "Actividad Subvencionada":"TALLER DIA DEL PADRE",
    "Importe en euros":"39.04 "
  },...
]
```

Trabajaremos con este fichero en Python para eliminar redundancias y estructurar mejor el documento. Para ello usaremos la biblioteca `json`, que consideraremos importada en el resto del capítulo como:

```
>>> import json
```

Las principales funciones proporcionadas por esta biblioteca son:

- `json.load(fich)`, que devuelve el objeto correspondiente (una lista o un diccionario) al JSON almacenado en el fichero `fich`. Análogamente, la función `json.loads(s)` analiza `s` (de tipo `str`, `bytes` o `bytearray`) y devuelve el objeto correspondiente.
- `json.dump(obj, fich)`, que escribe en el fichero `fich` la representación JSON del objeto `obj`, que debe ser una lista o un diccionario y contener valores a su vez representables en JSON. Análogamente, la función `json.dumps(obj)` devuelve una cadena de texto con la representación en JSON del objeto, si es posible.
- Resulta interesante ver algunos de los parámetros que pueden tomar todas estas funciones:
  - Podemos indicar la indentación con el parámetro `indent`, lo que facilitará la lectura especialmente al escribir en disco o transformar en cadena de texto; en nuestros ejemplos usaremos `indent=4`; en otro caso (por defecto su valor es `None`) obtendríamos todo el JSON en una única línea.
  - Dado que las parejas en un objeto no tienen orden, podemos indicar si queremos que Python las ordene alfabéticamente con `sort_keys=True` (por defecto su valor es `False`).
  - Por último, cuando estamos trabajando con valores con tildes, es útil usar el parámetro `ensure_ascii=False` (por defecto `True`) .

Vamos a cambiar la estructura del documento, de tal manera que pasemos a tener una lista de objetos, cada uno de los cuales consta de dos parejas, una indicando el nombre de la asociación y otra una lista `Actividades` de objetos que contengan el nombre de cada actividad subvencionada y su importe. Así, evitaremos repetir en cada objeto el nombre de la asociación y tendremos en un mismo objeto la información sobre cada uno de los centros. A continuación, mostramos cómo quedaría nuestro fichero después de la transformación:

```
[
    { "Asociación": "AMPA ANTONIO MACHADO",
        "Actividades": [
            {"Actividad Subvencionada": "TALLER FIESTA DE
              CARNAVAL",
             "Importe en euros": "94.56"
            },
                ...
            {"Actividad Subvencionada": "SAN ISIDRO",
             "Importe en euros": "195.00 "
            }
        ]
    },
    ...
]
```

En este caso, los primeros puntos suspensivos indican que hay más actividades para esa asociación, mientras que los segundos indican que hay más asociaciones. Para implementar esa transformación empezamos abriendo los ficheros correspondientes, esta vez con codificación `utf-8`, y usando la función `load` para cargar en `datos` una lista con la información del fichero.

```
>>> with open(ruta1, encoding='utf-8') as fich_lect,
open(ruta2, 'w', encoding='utf-8') as fich_escr:
>>> datos = json.load(fich_lect)
```

A continuación, definimos algunas constantes para asegurarnos de no equivocarnos al escribir cadenas que aparecen repetidas veces e inicializamos las variables que vamos a usar. Son particularmente interesantes las variables `lista`, que almacena los valores que formarán parte del *array* principal del JSON, y `dicc`, que almacena el diccionario que posteriormente servirá como objeto de cada una de las asociaciones. Por su parte, `lista_act` acumula las actividades para la asociación en la que estamos trabajando en el momento actual:

```
>>>     asoc_str = "Asociación"
>>>     act_str = "Actividad Subvencionada"
>>>     imp_str = "Importe en euros"
>>>     lista = []
>>>     list_act = []
>>>     asoc_actual = ""
>>>     dicc = {}
```

Procedemos ahora a recorrer todos los elementos del JSON, extrayendo sus valores:

```
>>>     for elem in datos:
>>>        asoc = elem[asoc_str]
>>>        act = elem[act_str]
>>>        imp = elem[imp_str]
```

Cuando cambiamos de asociación almacenamos las actividades acumuladas en `list_act` en la asociación actual, vaciamos la lista de actividades y creamos un nuevo diccionario para la nueva actividad, que metemos en la lista:

```
>>>        if asoc_actual != asoc:
>>>            dicc["Actividades"] = list_act
>>>            list_act = []
>>>            dicc = {"Asociación": asoc}
>>>            lista.append(dicc)
```

En todo caso, añadimos la información sobre las actividades en el paso actual y actualizamos el nombre de la asociación con la que estamos trabajando:

```
>>>        list_act.append({act_str : act,
                            imp_str : imp})
>>>        asoc_actual = asoc
```

Por último, usamos `dump()` para escribir en disco el resultado almacenado en `lista`, usando opciones de formato para que el fichero pueda ser leído fácilmente.

```
>>>     json.dump(lista, fich_escr, ensure_ascii=False,
                  indent=4)
```

Dado que JSON nos permite añadir información a varios niveles, podríamos añadir el importe total de las subvenciones recibidas por una asociación. Es sencillo modificar el algoritmo que hemos mostrado anteriormente para hacerlo, pero al ejecutarlo encontramos un error en el código, en particular en la línea en la que transformamos de cadena a número el importe de cada actividad:

```
>>> imp = float(elem[imp_str])
```

Al analizar cuidadosamente el fichero JSON encontramos que algunos importes tienen la forma `1.000.00`, con un punto para indicar los millares y otro para indicar la parte decimal. Este error no aparecía anteriormente porque nos limitamos a copiar valores, que al leer eran simplemente cadenas, pero al estar interesados en sumar la transformación a número de Python falla. ¿Cómo podemos arreglar este problema? Dado que tenemos la misma información en distintos formatos (ya hemos visto que la tenemos en CSV, TSV y Excel, además de XML, como mostraremos en la siguiente sección) podemos usar uno de estos ficheros para extraer la información que necesitamos. El código necesario para transformar la información en CSV en el JSON

formateado que hemos propuesto es muy similar al código mostrado arriba, ya que de hecho ambos trabajan con una lista de diccionarios; en el repositorio del libro está disponible la función completa.

XML

XML, del inglés *eXtensible Markup Language* (lenguaje de marcado extensible), es un lenguaje de marcado que nos permite almacenar información estructurada en forma de árbol. Entenderemos que cada fragmento de información es un *elemento* y tendrá una *etiqueta* asociada. Para una etiqueta etq, escribiremos la información del elemento entre las *marcas* `<etq>Información</etq>`, donde `<etq>` es la marca de inicio, `</etq>` es la marca de fin, e `Información` son los datos que queremos almacenar, que pueden ser otros elementos o directamente valores.

Por ejemplo, si queremos guardar información de una persona podemos pensar en los elementos persona, nombre, apellidos y edad, todos ellos representados con etiquetas con dichos nombres. Por tanto, Fulanito García, de 18 años, se almacenaría en XML como:

```
<persona>
   <nombre>Fulanito</nombre>
   <apellidos>García</apellidos>
   <edad>18</edad>
</persona>
```

Además, los elementos pueden tener *atributos* dentro de la marca de inicio, indicando propiedades del elemento. Por ejemplo, si queremos añadir el identificador como propiedad de persona, podríamos usar un atributo `dni`. Suponiendo que el DNI de Fulanito García es 12345a, cambiaríamos la marca de inicio a `<persona dni="12345a">`. Como identificador del atributo usaremos una cadena sin espacios, mientras que el valor siempre estará encerrado entre comillas.

Para que un fichero XML esté bien formado es necesario que contenga un único elemento principal, que a su vez puede contener varios. Es decir, si deseamos almacenar una lista de elementos necesitamos crear un elemento raíz que los contenga a todos. Además, es posible definir *ficheros DTD* (*Document Type Definition*, definición de tipo de documento) y *XML Schemas* (esquemas XML) indicando los atributos, los subelementos y el tipo de la información almacenada en cada elemento. Este tipo de documento queda fuera de los objetivos del presente libro, por lo que simplemente supondremos que los documentos XML que analizamos están bien formados.

Ahora que conocemos cómo es un fichero XML, ¿qué estructura de datos es la más adecuada para representarlo en Python? En general la respuesta es un *árbol*, donde el elemento único del documento será la raíz, cada elemento se corresponderá con un nodo que tendrá por hijos los elementos que contiene y como atributos su valor y sus atributos, estos últimos habitualmente representados como un diccionario. Existen en Python varias bibliotecas que siguen esta aproximación y que se distinguen por aspectos concretos de su implementación, como es la eficiencia de algunas operaciones o su robustez frente a distintos ataques[2]. Nosotros presentamos a continuación la biblioteca `etree`, la biblioteca estándar de Python, que es robusta frente a la mayoría de ataques, y presenta una interfaz lo bastante general que permitirá a los lectores entender otras bibliotecas similares, en el caso de necesitarlo.

La biblioteca `etree` está compuesta por dos clases principales, `ElementTree`, usada para representar el árbol completo, y `Element`, usada para representar los nodos. Los principales métodos de la clase `ElementTree` son:

- La constructora `ElementTree()`, que devuelve un árbol vacío.
- El método `parse(ruta)`, que devuelve el objeto de clase `ElementTree` correspondiente al fichero en la `ruta`.
- El método `write(ruta)`, que almacena en la ruta indicada el árbol.
- El método `getroot()`, que devuelve el objeto `Element` correspondiente a la raíz del árbol.
- El método `_setroot(nodo)`, que fija como raíz del árbol el `nodo` dado como argumento.
- El método `iter(tag)`, que devuelve un iterador sobre los nodos del árbol, siguiendo un recorrido de primero en profundidad empezando desde la raíz. El iterador recorre aquellos nodos que coincidan con la etiqueta `tag`; si la función no recibe ningún argumento su valor por defecto es `None` y el iterador recorrerá todos los nodos.
- El método `findall(patron, namespace)`, que devuelve una lista con todos aquellos nodos, a partir de la raíz, que encajen con `patron` (bien una etiqueta o una ruta de etiquetas) en el espacio de nombres dado (por defecto `None`). Análogamente, la función `find(patron, namespace)` devuelve el primer nodo que cumple las condiciones. En el capítulo 2 veremos XPath, un método más potente para buscar elementos en documentos HTML.

[2] Véase `https://docs.python.org/3/library/xml.html#xml-vulnerabilities` para más detalles.

Por su parte, los principales métodos y atributos de la clase `Element` son:

- La constructora `Element(etq)`, que construye un nodo con la etiqueta `etq`.
- La constructora `SubElement(padre, etq)`, que construye un nodo como hijo de `padre` y con etiqueta `etq`.
- El método `fromstring(cadena)`, que devuelve la raíz del árbol representado en `cadena`.
- El atributo `tag`, que almacena la etiqueta del nodo.
- El atributo `text`, que almacena el valor del nodo.
- El atributo `attrib`, que almacena el diccionario con los atributos.
- El método `iter(tag)`, que se comporta de manera análoga al método del mismo nombre para `ElementTree`, pero actuando desde el nodo que hace la llamada.
- Los métodos `findall(patron, namespace)` y `find(patron, namespace)`, que se comportan igual que los métodos del mismo nombre para `ElementTree`, pero actuando desde el nodo que hace la llamada.
- Es interesante observar que podemos iterar sobre un elemento, con lo que recorreremos todos los hijos directos. También es posible acceder a los elementos por debajo del nodo con notación de listas. Así, el primer hijo de un nodo será `nodo[0]`.

Ahora que conocemos la estructura de los ficheros XML y las clases principales de la biblioteca `etree`, que supondremos cargada como

```
import xml.etree.ElementTree as ET
```

pasamos a analizar cómo ha representado el Gobierno las subvenciones. Análogamente a lo que sucedía con JSON, el fichero no aprovecha la flexibilidad de XML y consiste en una lista de elementos `Row` (fila), con 3 elementos cada uno que se corresponden con las filas de los ficheros CSV y Excel:

```
<Root>
      <Row>
            <Asociaci_n>AMPA ANTONIO MACHADO</Asociaci_n>
            <Actividad_Subvencionada>TALLER FIESTA DE
              CARNAVAL</Actividad_Subvencionada>
            <Importe>94.56</Importe>
      </Row>
      ...
</Root>
```

donde vemos que los creadores del documento intentaron usar etiquetas con tildes pero no usaron el formato adecuado, por lo que la primera etiqueta de cada fila

aparece como `Asociaci_n`. Explicamos en primer lugar cómo calcular el diccionario que relaciona asociaciones e importe total que hemos calculado para formatos anteriores. Empezaremos obteniendo el árbol desde el fichero con la función `parse`, extraeremos la raíz para iterar sobre ella e inicializamos el diccionario acumulador:

```
>>> arbol = ET.parse(ruta)
>>> raiz = tree.getroot()
>>> asocs = {}
```

Al iterar sobre la raíz entraremos en los elementos fila. Como sabemos el orden de los hijos podemos acceder de forma directa a ellos, obteniendo el nombre de la asociación en la posición `0` y el importe en la `2`:

```
>>> for fila in raiz:
>>>     centro = fila[0].text
>>>     subvencion = float(fila[2].text)
>>>     if centro in asocs:
>>>         asocs[centro] = asocs[centro] + subvencion
>>>     else:
>>>         asocs[centro] = subvencion
>>> print(asocs)
```

Vamos ahora cómo escribir un nuevo fichero. Como en el caso de JSON, proponemos una estructura que presente una lista de asociaciones, para cada una de las cuales tendremos un atributo para el nombre, un elemento para almacenar una lista de parejas actividad-importe, y un segundo elemento con el importe total en subvenciones obtenido por la asociación. Así pues, nuestro objetivo es obtener un fichero con la estructura:

```
<Raiz>
  <Asociacion nombre="AMPA ANTONIO MACHADO">
    <Actividades>
      <Actividad>
        <Nombre>TALLER FIESTA DE CARNAVAL</Nombre>
        <Subvencion>94.56</Subvencion>
      </Actividad>
      ...
      <Actividad>
        <Nombre>SAN ISIDRO</Nombre>
        <Subvencion>195.0</Subvencion>
      </Actividad>
    </Actividades>
    <Total>2344.99</Total>
  </Asociacion>
  ...
</Raiz>
```

Empezamos de nuevo cargando el árbol con la función `parse` y accediendo a la raíz:

```
>>> arbol = ET.parse(ruta1)
>>> raiz = arbol.getroot()
```

Por su parte, crearemos el nuevo árbol con la constructora `ElementTree`, crearemos un nuevo nodo raíz con la constructora `Element` y lo fijaremos con `_setroot`.

```
>>> nuevo = ET.ElementTree()
>>> raiz_nueva = ET.Element("Raiz")
>>> nuevo._setroot(raiz_nueva)
```

A partir de ahora nuestro objetivo será crear elementos para cada asociación y añadirles a su vez como elementos las actividades y el importe total. Creamos así elementos acumuladores como sigue:

```
>>> elem_actual = ET.Element("Asociacion")
>>> asoc_actual = ""
>>> actividades = ET.SubElement(elem_actual, "Actividades")
>>> gasto = 0
```

Pasamos a recorrer todos los elementos `Row` de la raíz con `findall` (también podríamos haber iterado sobre la raíz, como hicimos anteriormente). Para cada elemento extraemos el nombre de la asociación y de la actividad y el importe:

```
>>> for fila in raiz.findall('Row'):
>>>     asoc = fila.find('Asociaci_n').text
>>>     act = fila.find('Actividad_Subvencionada').text
>>>     imp = float(fila.find('Importe').text)
```

Cuando cambiamos de asociación debemos usar la constructora `SubElement` para añadir a los elementos correspondientes la información acumulada y reinicializar las variables:

```
>>>     if asoc_actual != asoc:
>>>         gas_total = ET.SubElement(elem_actual, "Total")
>>>         gas_total.text = str(gasto)
>>>         elem_actual = ET.SubElement(raiz_nueva, "Asociacion")
>>>         elem_actual.set('nombre', asoc)
>>>         actividades = ET.SubElement(elem_actual, "Actividades")
>>>         gasto = 0
```

Durante todas las iteraciones del bucle insertamos en el elemento para actividades información sobre la actividad actual y su importe, además de actualizar el acumulador para el gasto:

```
act_elem = ET.SubElement(actividades, "Actividad")
nom_elem = ET.SubElement(act_elem, "Nombre")
nom_elem.text = act
imp_elem = ET.SubElement(act_elem, "Subvencion")
imp_elem.text = str(imp)
gasto = gasto + imp
asoc_actual = asoc
```

Una vez acabamos de recorrer el fichero, grabamos en disco usando `write`.

```
>>> nuevo.write(ruta2)
```

CONCLUSIONES

En este capítulo hemos visto que es posible cargar y manipular en Python distintos tipos de ficheros. Sin embargo, buena parte del trabajo nos "venía hecho", ya que los ficheros estaban en general para ser leídos. ¿Habrá ficheros listos para descargar sobre cualquier tema que necesitemos? Obviamente no, en general deberemos crear nuestros propios ficheros a partir de información más o menos estructurada que encontramos en la web. En el próximo capítulo veremos cómo ciertas páginas, que contienen demasiada información como para presentarla en ficheros descargables, ofrecen a los programadores una serie de funciones para descargar los datos de manera sencilla.

REFERENCIAS

- Kenneth Alfred Lambert. Fundamentals of Python: First programs. CENGAGE Learning (segunda edición), 2017.
- Mark J. Johnson. A concise introduction to programming in Python. Chapman & Hall (segunda edición), 2018.
- Documentación del módulo `csv` (accedida el 15 de marzo de 2018). `https://docs.python.org/3/library/csv.html`.
- Documentación del módulo `xlrd` (accedida el 16 de marzo de 2018). `http://xlrd.readthedocs.io/en/latest/`.
- Documentación del módulo `xlwt` (accedida en marzo de 2018). `http://xlwt.readthedocs.io/en/latest/`.
- Documentación de la clase `ElementTree` (accedida en marzo de 2018). `https://docs.python.org/3/library/xml.etree.elementtree.html`.
- Documentación del módulo `json` (accedida en marzo de 2018). `https://docs.python.org/3/library/json.html`.

2 WEB SCRAPING

INTRODUCCIÓN

La Web es una fuente inagotable de información. Blogs, foros, sitios oficiales que publican datos de interés… todos ofrecen información que, recopilada de forma concienzuda, puede sernos de gran utilidad: evolución y comparativa de precios en tiendas *online*, detección de eventos, análisis de la opinión de usuarios sobre productos, crear *chatbots,* etc.

El límite es el que ponga nuestra imaginación, aunque siempre, por supuesto, dentro de la legislación vigente. Debemos recordar que algunos de estos datos pueden estar protegidos por derechos de autor, patentes, etc., así que es nuestra responsabilidad ser cuidadosos y consultar las posibilidades que ofrece la página. Además, muchos sitios web pueden llegar a bloquear nuestra IP si detectan que estamos accediendo a sus datos sin permiso.

En este sentido, hay que mencionar el concepto de *datos abiertos*, adoptado cada vez más por organizaciones privadas y públicas, que busca proporcionar datos accesibles y reutilizables, sin exigencia de permisos específicos y que logra que cada día tengamos más datos disponibles.

En todo caso, el problema es: ¿cómo “capturar” esta información?

Las páginas pueden ofrecernos información en 3 formatos principales.

La primera posibilidad es que las propias webs incluyan ficheros ya preparados en alguno de los formatos vistos en el capítulo 1 en forma de ficheros que se pueden descargar directamente. En tales casos, bastará con descargar estos ficheros (en seguida veremos cómo) para poder analizarlos.

La segunda posibilidad, muy común en las webs preparadas para consultas, es que el propio sitio web ofrezca acceso a sus datos a través de un protocolo API-REST, al que podremos hacer peticiones sobre datos concretos, que obtendremos en formato JSON o XML. Hablaremos de esta posibilidad en el capítulo siguiente.

Finalmente, la tercera posibilidad es que la información forme parte de la propia página web, en un formato muy cómodo para los usuarios "humanos" que la visitan, pero poco tratable para un programa. Por fortuna, Python incluye bibliotecas que nos ayudarán a extraer esta información, es lo que se conoce propiamente con el término *web scraping*.

Dentro de esta última posibilidad, la de tener los datos incrustados como contenido de la propia página web, tenemos a su vez dos posibilidades. En la primera, la información está disponible simplemente accediendo a la URL (dirección web) de la página. En este caso, tras descargar el contenido de la página, emplearemos una biblioteca como `BeautifulSoup`, que nos permita buscar la información dentro del código de la página.

En otros casos, cada vez más frecuentes, la página requerirá información por nuestra parte antes de mostrarnos los datos. Aunque sea más complicado, también podremos hacerlo mediante bibliotecas como `selenium`, que nos permite hacer desde un programa todo lo que haríamos desde el navegador web.

La siguiente tabla muestra esta división, que también corresponde a la estructura de este capítulo:

Fuentes Web de datos	Biblioteca Python
Ficheros incluidos en la página web	`requests, csv…`
API-REST	`Requests`
Datos que forman parte de la página	`BeatifulSoup`
Datos que requieren interacción	`Selenium`

FICHEROS INCLUIDOS EN LA PÁGINA WEB

En ocasiones las organizaciones nos ofrecen los datos ya listos para descargar, con formatos como los que hemos visto en el capítulo uno. En estos casos lo más sencillo es utilizar una biblioteca como `requests`, que permite descargar el fichero en el disco local y tratarlo a continuación, o bien, si el fichero es realmente grande hacer un tratamiento en *streaming*, sin necesidad de mantener una copia local.

URIs, URLs y URNs

Estas tres palabras aparecen a menudo cuando se habla de páginas web, e interesa que tengamos una idea precisa de su significado.

Una URI (*Uniform Resource Identifier* o identificador de recursos uniforme) es un nombre, o formalmente una cadena de caracteres, que identifica un recurso de forma accesible dentro de una red: una página web, un fichero, etc. Su forma genérica es:

schema:[**//**[**user**[**:passwd**]**@**]**host**[**:port**]][**/path**][?**query**][#**tag**]

donde las partes entre corchetes son opcionales. El esquema inicial indica la "codificación" de la URI por ejemplo "`http:`". Después vienen, opcionalmente, un nombre de usuario y su palabra clave, seguidos de un nombre del *host*, el ordenador al que nos conectamos, o su correspondiente IP y un puerto de acceso. A toda esta parte de la URI con forma (**//**[**usuario**[**:passwd**]**@**]**host**[**:port**]) se la conoce como *autoridad*. Después viene la ruta (*path*), que es una secuencia de nombres separadas por ya sea por los símbolos "`/`" o "`:`". Finalmente, puede incluirse una posible consulta (*query*) y una etiqueta (*tag*) para acceder a un lugar concreto del documento. Un ejemplo de URI sería:

```
https://es.wikipedia.org/wiki/Alan_Turing#Turing_en_el_cine
```

En esta URI, el esquema viene dado por el protocolo `https`, la autoridad corresponde con `es.wikipedia.org` (no hay usuario ni palabra clave) y la ruta es la cadena `wiki/Alan_Turing`. La URI no contiene consultas, pero sí una etiqueta final, tras el símbolo "#": `Turing_en_el_cine`. Para ver ver URIs que incluyan consultas basta con que nos fijemos en la dirección que muestra nuestro navegador cuando hacemos una búsqueda en Google o en YouTube; allí está nuestra consulta tras el símbolo "?" y normalmente en forma de parejas *clave=valor*, separadas por los símbolos "&" o "`:`".

Por ejemplo `https://www.youtube.com/watch?v=iSh9qg-2qKw`. En este caso, la consulta se incluye con la forma *v=codificaciónDeLaConsulta*.

Las URIs se suelen dividir en dos tipos: URLs y URNs. Las URLs (*Uniform Resource Locator* o localizador de recurso uniforme), a menudo conocidas simplemente como *direcciones web*, sirven para especificar un recurso en la red mediante su localización y una forma o protocolo que permite recuperarlo, que corresponde con la parte *schema* de la estructura general de la URI que hemos mostrado anteriormente. La barra de las direcciones de nuestro navegador normalmente muestra URLs. Los ejemplos de URIs que hemos visto en los párrafos anteriores son, en particular, URLs. Además de esquemas o protocolos `http` o `https`, en ocasiones encontraremos URLs que especifican otros, como el *File Transfer Protocol* (FTP), por ejemplo en la siguiente URL: `ftp://example.com`.

Para complicar un poco la cosa, hay que mencionar que existen páginas web que incluyen algunos caracteres como "`[`" o "`]`" que no están contemplados en la definición técnica de URI, pero sí en la de URL, así que serían URLs pero no URIs. A pesar de estos casos excepcionales, no debemos olvidar que la idea general y "oficial" es que las URLs son tipos particulares de URIs.

Finalmente, una `URN` (*Uniform Resource Name*, o nombre de recurso uniforme), el otro tipo de URIs, identifica un recurso, pero no tienen por qué incluir ninguna forma de localizarlos. Un ejemplo puede ser `urn:isbn:0-391-31341-0`.

Dado que en este capítulo estamos interesados en acceder a recursos, y por tanto necesitaremos su localización, usaremos a partir de ahora únicamente URLs.

Ejemplo: datos de contaminación en Madrid

`http://www.mambiente.munimadrid.es/opendata/horario.txt`

Esta URL proporciona los datos de las estaciones de medición de la calidad del aire de la ciudad de Madrid. Supongamos que queremos descargar el fichero correspondiente para analizarlo a continuación. Para ello, nos bastará con incluir la biblioteca `requests` que incluye un método para "pedir" (`get`) el fichero, que podemos grabar a continuación en nuestro disco duro local:

```
>>> import requests
>>> url = "http://www.mambiente.munimadrid.es/opendata/horario.txt"
>>> resp = requests.get(url)
>>> print(resp)
```

Tras "bajar" el fichero, con `get`, mostramos con `print` la variable `resp` que nos mostrará por pantalla el llamado *status_code*, que nos informa del resultado de la descarga:

```
<Response [200]>
```

El valor `200` indica que la página se ha descargado con éxito. En la URL:

```
https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
```

podemos encontrar información detallada acerca de los distintos *status_code*, pero a nuestros efectos baste con decir que los que empiezan con 2 suelen indicar éxito, mientras que los que comienzan por 4 o 5 indican error, como el famosísimo 404 (recurso no encontrado) o el 403 (acceso prohibido al recurso).

Por comodidad vamos a grabar el fichero descargado en un archivo local, al que llamaremos `horario.txt`. El contenido de la página está en el atributo `content` de la variable `resp`:

```
>>> path = '…carpeta de nuestro disco local…'
>>> with open(path + 'horario.txt', 'wb') as output:
       output.write(resp.content)
```

Antes de ejecutar este fragmento de código, debemos reemplazar la variable `path` por una dirección en nuestro disco local. La construcción Python `with` asegura que el fichero se cerrará automáticamente al finalizar.

Ya tenemos el fichero en nuestro ordenador y podemos tratarlo. En nuestro ejemplo el fichero contiene los datos de contaminación del día en curso. El formato es:

- Columnas 0, 1, 2: concatenadas identifican la estación. Por ejemplo 28,079,004 identifica a la estación situada en la Plaza de España.
- Columnas 3, 4, 5: identifican el valor medido. Por ejemplo, si la columna 3 tiene el valor 12 indica que se trata de una medida de óxido de nitrógeno (las otras dos columnas son datos acerca de la medición que no vamos a usar).
- Columnas 6, 7, 8: año, mes, día de la medición, respectivamente.
- Columnas 9 – 56: indican el valor en cada hora del día. Van por parejas, donde el primer número indica la medición y el segundo es "V" si es un valor válido o "N" si no debe tenerse en cuenta. Por tanto, la columna 9 tendrá la medición a las 00 horas si la columna 10 es una "V", la columna 11 la medición a las 01 horas si la columna 12 es una "V", y así sucesivamente. En la práctica, el primer valor "N" corresponde con la hora actual, para la que todavía no hay medición.

Podemos usar esta información para mostrar los datos en formato de gráfica que indique la evolución de la contaminación por óxido de nitrógeno en un día cualquiera a partir de los datos de la estación situada en la Plaza de España de Madrid.

Para ello comenzamos por importar la biblioteca `csv`, ya que se trata de un fichero de texto separado por comas, y la biblioteca `matplotlib.pyplot` que nos permite hacer gráficos de forma muy sencilla, y que trataremos en más detalle en un capítulo posterior:

```
>>> import matplotlib.pyplot as plt
>>> import csv
```

Ahora abrimos el fichero y generamos el vector con los valores que corresponden a la línea de la Plaza de España para el óxido de nitrógeno. Lo que hacemos es recorrer las columnas por su posición, primero para seleccionar la línea que corresponde a la estación de la Plaza de España y para el valor 12 (óxido de nitrógeno), y, una vez localizada la línea con los datos, recorrer de la columna 9 en adelante añadiendo valores al vector `vs` hasta encontrar la primera "N" en la columna siguiente al valor:

```
>>> with open(path + 'horario.txt') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    for row in readCSV:
        if (row[0]+row[1]+row[2]=='28079004' and
            row[3]=='12'):
            plt.title("Óxido de nitrógeno: "
                      +row[8]+"/"+row[7]+"/"+row[6])
            hora = 0
            desp = 9
            vs = []
              horas = []
            while hora<=23:
              if row[desp+2*hora+1]=='V':
                vs.append(row[desp+2*hora])
                horas.append(hora)
              hora +=1
            plt.plot(horas, vs)
            plt.show()
```

La variable `desp` indica la columna dentro de cada fila del fichero en la que empiezan los valores medidos. En particular, para cada hora, la columna `desp+2*hora` contiene el valor medido en esa hora, que solo será válido si la columna `desp+2*hora+1` contiene una "V". Tras recoger los valores válidos en el array `vs` y sus horas asociadas en el array `horas`, la llamada a `plot` crea la gráfica a

partir de dos vectores: uno con los valores *x*, que en este caso son las horas y otro, de la misma longitud, con los valores *y*, en este caso los valores medidos.

La figura 2-1 nos muestra un ejemplo para un día de diario. Vemos la subida en el nivel de contaminación en la hora punta de entrada, y un repunte, aunque de menor intensidad, a la hora de comer.

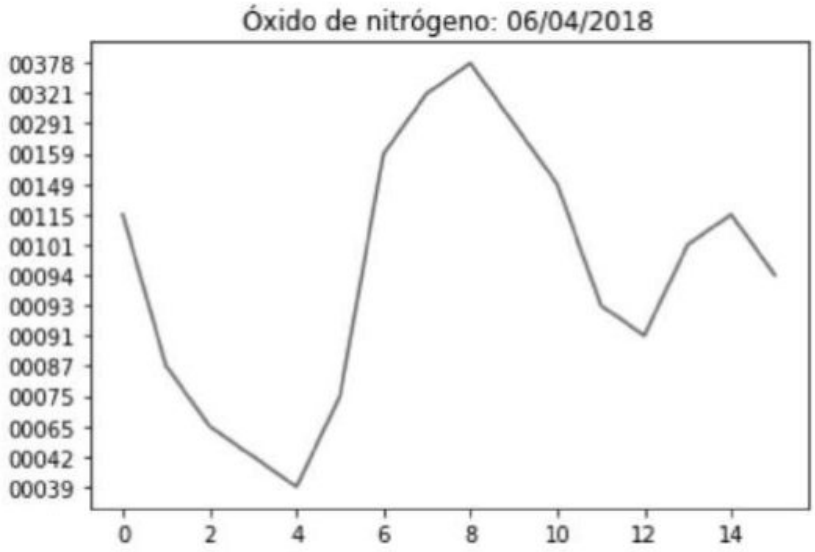


Figura 2-1. Evolución del óxido de nitrógeno entre las 0 y las 14h.

De esta forma, guardando los ficheros correspondientes a cada día podríamos calcular medias anuales, ver la evolución de la contaminación, o incluso intentar predecir los valores por anticipado.

Aunque en este caso el fichero es de reducido tamaño, en ocasiones los ficheros sobre los que queremos trabajar pueden ser realmente grandes, y puede que no nos interese descargarlos completos. En estos casos, podemos utilizar el procesamiento *perezoso*, que busca utilizar la menor información posible. Inicialmente, importamos las bibliotecas necesarias:

```
>>> import requests
>>> from contextlib import closing
>>> import csv
>>> import codecs
>>> import matplotlib.pyplot as plt
```

Llama la atención la incorporación de dos nuevas bibliotecas:

- `codecs`: utilizada para leer directamente los *strings* en formato `utf-8`.

- `contextlib`: nos permite leer directamente el valor devuelto por *`requests.get`*.

Ahora el resto del código:

```
>>> url = "http://www.mambiente.munimadrid.es/opendata/horario.txt"
>>> with closing(requests.get(url, stream=True)) as r:
    reader = csv.reader(codecs.iterdecode(
                        r.iter_lines(),
                        'utf-8'),
                       delimiter=',')
    for row in reader:
        ..... # igual que en el código anterior
```

Este código hace lo mismo que el anterior, pero evita:

a) Tener que grabar el fichero en disco, gracias a la lectura directa de `request.get()`.
b) Cargarlo completo en memoria, gracias a la opción `stream=True` de `requests.get()`.

DATOS QUE FORMAN PARTE DE LA PÁGINA

El siguiente método consiste en obtener los datos a partir de una página web. Para extraer los datos, necesitaremos conocer la estructura de la página web, que normalmente está compuesta por código *HTML*, imágenes, *scripts* y ficheros de estilo. La descripción detallada de todos estos componentes está más allá del propósito de este libro, pero vamos a ver los conceptos básicos que nos permitan realizar nuestro objetivo: hacer *web scraping*.

Lo que oculta una página web

Lo que vemos en el navegador es el resultado de interpretar el código HTML de la página. Veamos un ejemplo de una página mínima en HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      Un mini ejemplo
    </title>
  </head>
```

```
  <body>
    <div id="date"> Fecha 25/03/2035 </div>
    <div id="content"> Un poco de texto </div>
  </body>
</html>
```

Lo primero que vemos, es <!DOCTYPE html> que indica que se trata de un documento HTML, el estándar de las páginas web. A continuación, encontramos <html>, la etiqueta que marca el comienzo del documento, y que se cerrará al final del documento con </html>. En HTML, igual que en XML, los elementos tienen una estructura con la forma:

```
<elemento atributo="valor">Contenido</elemento>
```

El atributo no es obligatorio, y un mismo elemento puede incluir más de uno. Dentro del documento HTML, esto es entre la apertura <html> y el cierre </html> del elemento principal, encontramos siempre dos elementos:

- <head> … </head>: describe la cabecera del documento, como puede ser su título, el autor, etc.
- <body> … </body>: es el contenido en sí de la página, que es lo que nosotros deseamos examinar.

Dentro del elemento body habrá otros elementos, que dependerán de la página, que a su vez pueden contener otros elementos, y así sucesivamente. En nuestro pequeño ejemplo el cuerpo solo tiene dos frases, marcadas por las etiquetas y . Podemos grabar este código en un fichero con el nombre que deseemos, por ejemplo mini.html. Haciendo doble clic sobre el fichero se abrirá el navegador y veremos la (humilde) página.

Podemos cargar la página como un fichero de texto normal, y mostrarla mediante la biblioteca BeautifulSoup:

```
>>> from bs4 import BeautifulSoup
>>> url = r"c:\...\mini.html"
>>> with open(url, "r") as f:
    page = f.read()

>>> soup = BeautifulSoup(page, "html.parser")
>>> print(soup.prettify())
```

Para poder probar el ejemplo debemos incluir la ruta al fichero en la variable url. La variable soup contiene la página en forma de cadena de caracteres, que

convertimos en un formato interno estructurado usando la constructora `BeautifulSoup`, a la que indicamos como segundo elemento que se debe emplear el analizador sintáctico (`parser`) propio de HTML, en este caso "`html.parser`". Si la página es muy compleja podemos necesitar otro analizador, como "`html5lib`", más lento en el procesamiento, pero capaz de tratar páginas más complicadas.

En nuestro, ejemplo el resultado del análisis se guarda en la variable `soup`. La última instrucción muestra el código HTML en un formato legible.

Un poco de HTML

Para hacer *web scraping* debemos identificar los elementos fundamentales que se pueden encontrar en un documento HTML. La siguiente lista no es, ni mucho menos, exhaustiva, pero sí describe los más habituales.

ELEMENTOS DE FORMATO

…: el texto dentro del elemento se escribirá en negrita. Por ejemplo, negrita escribirá **negrita**.

…: el texto dentro del elemento se escribirá en *itálica*.

<h1>…</h1>: el texto se escribe en una fuente mayor, normalmente para un título. También existen <h2>…</h2>, <h3>…</h3> y así hasta <h6> …</h6> para títulos con fuente menor.

… : un nuevo párrafo.

: salto de línea. De los pocos elementos sin etiqueta de apertura y de cierre.

<pre>…</pre>: preserva el formato, muy útil por ejemplo para escribir código sin temor a que algún fragmento sea interpretado como un elemento HTML.

  : un espacio en blanco "duro", indica al navegador que en ese punto no puede romper la línea.

… : permite crear secciones dentro del texto con un estilo común. Suelen llevar un atributo *class* que identifica el estilo dentro de una hoja de estilos.

…: similar a … pero suele especificar el estilo dentro del propio elemento. Por ejemplo, para poner un texto en rojo se puede escribir: *¡alerta!*

LISTAS

**
…:** Lista sin orden. Los elementos se especifican con …. Por ejemplo:

```
<ul>
<li>Uno</li>
<li>Dos</li>
<li>Tres</li>
</ul>
```

Mostrará:

- Uno
- Dos
- Tres

**
…
**: lista ordenada. Por ejemplo:

```
<ol>
<li>Uno</li>
<li>Dos</li>
<li>Tres</li>
</ol>
```

Mostrará:

1. Uno
2. Dos
3. Tres

ENLACES

…: Especifica un fragmento de texto o una imagen sobre la que se puede hacer clic. Al hacerlo, el navegador "saltará" a la dirección especificada por `href`, que puede ser bien una dirección web o un marcador en la propia página. Por ejemplo <a href:"https://rclibros.es/"> *Pulsa aquí* .

IMÁGENES

: Incluye en la página la imagen indicada en `src`. Se suele emplear junto con los atributos `width` y/o `height` para reescalar la imagen. Si solo se incluye uno de los dos, el otro reescalará proporcionalmente. Por ejemplo

```
<img src = "/html/images/test.png" alt="Test" width = 300 />
```

mostrará la imagen `test.png` que debe estar en la carpeta `images`, con una anchura de 300 puntos. El atributo `alt="Test"` contiene un texto que se mostrará en caso de que la imagen no se encuentre en el lugar especificado.

TABLAS

`<table>…</table>`: Las tablas son estructuras que encontraremos a menudo al hacer *web scraping*, y conviene conocer su estructura de 3 niveles:

1. El primer nivel es el de la propia tabla y viene delimitado por los *tags* `<table>…</table>`. A menudo se incluyen atributos como `border`, que especifica la forma del borde de la tabla, `cellpadding`, que indica el número de píxeles que debe haber como mínimo desde el texto de una celda o casilla hasta el borde, y `cellspacing`, que indica el espacio entre una celda y la siguiente.

2. El segundo nivel corresponde a las filas, cada fila se delimita por `<tr> …</tr>`.

3. Finalmente, el tercer elemento es el nivel de celda o casilla, y viene delimitado por `<td>…</td>`, o en el caso de ser las celdas de cabecera por `<th>…</th>`.

Un ejemplo nos ayudará a entender mejor esta estructura:

```
<table border = "1">
   <tr>
      <td>fila 1, columna 1</td>
      <td>fila 1, columna 2</td>
   </tr>

   <tr>
      <td>fila 2, columna 1</td>
      <td>fila 2, columna 2</td>
   </tr>
</table>
```

Mostrará:

fila 1, columna 1	fila 1, columna 2
fila 2, columna 1	fila 2, columna 2

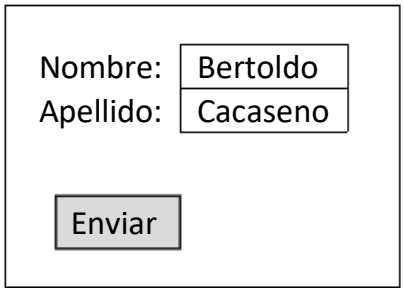
FORMULARIOS

`<form>…</form>`: Los formularios se utilizan para que el usuario pueda introducir información que será posteriormente procesada. En *web scraping* son muy importantes, porque a menudo encontraremos páginas que solo nos muestran la información como respuesta a los valores que debemos introducir nosotros (por ejemplo, tenemos que introducir previamente un usuario y una contraseña).

Los formularios ofrecen muchas posibilidades, permitiendo definir listas desplegables, varios tipos de casillas para marcar, etc. Aquí solo vamos a ver un pequeño ejemplo para conocer su aspecto general. Se trata de una sencilla página en la que va a pedir dos valores de texto, un nombre y un apellido, y tras pulsar en un botón de envío recogerá la información y la hará llegar a un módulo PHP (que no incluimos ya que escapa del ámbito de este libro).

```
<form action="/trata.php">
    Nombre:
    <input type="text" name="firstname" value="Bertoldo">
    <br>
    Apellido:
    <input type="text" name="lastname" value="Cacaseno">
  <br>
    <input type="submit" value="Enviar">
</form>
```

El aspecto de este formulario será el siguiente:



La parte fundamental y en la que debemos fijarnos son los elementos `<input>` (que, por cierto, no necesitan etiqueta de cierre `</input>`). El atributo `type` nos indica el tipo de entrada. El valor `text` de las dos primeras entradas del ejemplo, indica que se trata de un campo de texto, mientras que el tipo entrada `submit` nos indicará que se trata de un botón para enviar los datos.

ATRIBUTOS MÁS USUALES

Hay 4 atributos que podremos encontrar en la mayoría de elementos HTML:

- `id`: Identifica un elemento dentro de la página. Por tanto, los valores asociados no deben repetirse dentro de la misma página.
- `title`: indica el texto que se mostrará al dejar el cursor sobre el elemento. Por ejemplo `<h1 title="cabera principal> Python y Big Data </h1>` mostrará "cabecera principal" al situar el cursor sobre el texto "Python y Big Data".
- `class`: asocia un elemento con una hoja de estilo (CSS), que dará formato al elemento: tipo de fuente, color, etc.
- `style`: permite indicar el formato directamente, sin referirse a la hoja de estilo. Por ejemplo:
  `<p style = "font-family:arial; color:#FF0000;"> ...texto...</p>`

Con esta información básica sobre HTML, ya estamos listos para empezar a recoger información de las páginas web.

Navegación absoluta

Como acabamos de ver, los documentos HTML, al igual que sucede en XML, siguen una estructura jerárquica. Los elementos que están directamente dentro de otros se dice que son *hijos* del elemento contenedor y *hermanos* entre sí. En nuestro miniejemplo, los elementos `<div id="date">` y `<div id="content">` son hermanos entre sí, y ambos son hijos del elemento `<body>`. Además, `<div id="date">` tiene a su vez un hijo, que en esta ocasión no es otro elemento sino un texto.

La biblioteca `BeautifulSoup` nos permite utilizar estos conceptos para navegar por el documento buscando la información que precisamos. Como ejemplo inicial, podemos preguntar por los hijos del documento raíz, y su tipo para hacernos una idea de lo que vamos a encontrarnos:

```
>>> hijosDoc = list(soup.children)
>>> print([type(item) for item in hijosDoc])
>>> print(hijosDoc)
```

El primer print nos indica que hay 3 hijos:

- El primero, de tipo `bs4.element.Doctype`, corresponde a la primera línea.

- El segundo, de tipo `bs4.element.NavigableString` que, como vemos en el siguiente `print`, es tan solo el fin de línea que se encuentra entre línea inicial y la etiqueta `<html>`, y que se representa por `\n`.
- El tercer elemento, del tipo `bs4.element.Tag`, corresponde al documento HTML en sí mismo.

Ahora podemos seleccionar el tercer elemento (índice 2) y tendremos acceso al documento HTML en sí.

```
>>> html = hijosDoc[2]
>>> print(list(html.children))
```

Repitiendo el proceso podemos ver que `html` tiene 5 hijos, aunque tres corresponden a saltos de línea, y solo 2 nos interesan: `head` y `body`. De la misma forma podemos obtener los elementos contenidos en `head` y `body`, y repetir el proceso hasta analizar toda la estructura. La figura 2-2 muestra el árbol resultante, donde por simplicidad se han eliminado los valores `\n` a partir del segundo nivel.

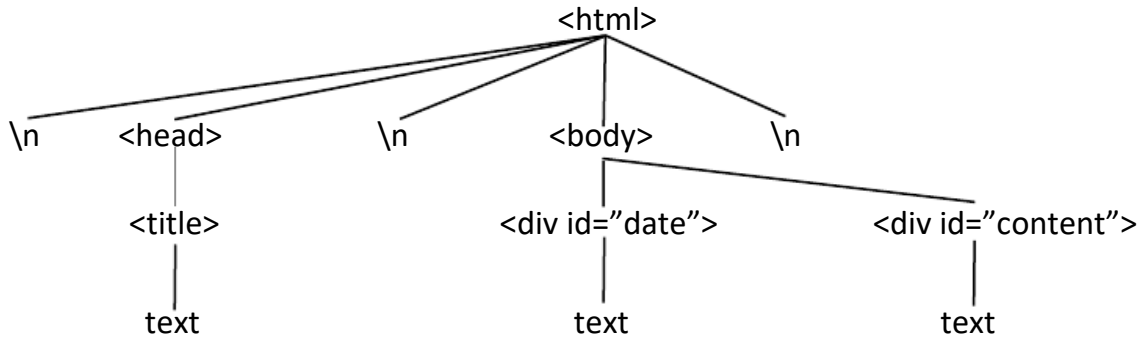


Figura 2-2. Árbol sintáctico de un documento.

En particular, supongamos que nos interesa el texto asociado al `div` con atributo `id = date`. Primero seleccionamos el elemento `body`, que es el cuarto hijo (índice Python 3) de `html`, y una vez más examinamos sus hijos:

```
>>> body = list(html.children)[3]
>>> print(list(body.children))
```

Comprobamos que `body` tiene 5 hijos: `\n`, el primer `div`, `\n`, el segundo `div`, y de nuevo `\n`. Como estamos interesados en el texto del primer `div`, accedemos a este elemento:

```
>>> divDate = list(body.children)[1]
```

Este elemento ya no tiene otros elementos tipo `tag` como hijos. Solo un elemento de tipo `bs4.element.NavigableString`, que corresponde al texto que buscamos. Podemos obtenerlo directamente con el método `get_text()`:

```
>>> print(divDate.get_text())
```

que muestra el valor deseado: `Fecha 25/03/2035`.

Navegación relativa

Para nuestro primer *web scraping* hemos recorrido el documento desde la raíz, es decir, hemos tenido que recorrer la *ruta absoluta* del elemento que buscamos. Sin embargo, en una página típica esto puede suponer un recorrido muy laborioso, implicando decenas de accesos al atributo *children*, una por nivel. Además, cualquier cambio en la estructura de la página que modifique un elemento de la ruta hará el código inservible.

Para evitar esto, *BeautifulSoup* ofrece la posibilidad de buscar elementos sin tener que explicar la ruta completa. Por ejemplo, el método `find_all()` busca todas las apariciones de un elemento a cualquier nivel y las devuelve en forma de lista:

```
>>> divs = soup.find_all("div")
```

Ahora, si sabemos que queremos mostrar el texto asociado al primer `div` podemos escribir:

```
>>> print(divs[0].get_text())
```

El resultado es el mismo valor (“`Fecha 25/03/2035`”), pero esta vez obtenido de una forma mucho más directa y sencilla. Podemos incluso reducir este código mediante el método `find`, que devuelve directamente el primer objeto del documento que representa el elemento buscado:

```
>>> print(soup.find("div").get_text())
```

Estos métodos siguen teniendo el problema de que continúan dependiendo, aunque en menor medida, de la ruta concreta que lleva hasta el elemento, ya que se basan en la posición del *tag* entre todos los del mismo tipo. Por ejemplo, si al creador de nuestra página se le ocurriera añadir un nuevo elemento `div` antes del que deseamos, sería este (el nuevo) el que mostraría nuestro código, ya que la posición del que buscamos habría pasado a ser la segunda (índice 1).

Para reducir esta dependencia de la posición, podemos afinar más la búsqueda apoyándonos en los valores de atributos usuales como `id` o `class`. En nuestro ejemplo, supongamos que sabemos que el elemento `div` que buscamos debe tener un `id` igual a `date`, y que este valor, como un `id`, es único en el código HTML. Entonces, podemos hacer la siguiente búsqueda, que esta vez es independiente del número de elementos `div` que pueda haber antes:

```
>>> print(soup.find("div", id="date").get_text())
```

`BeautifulSoup` incluye muchas más posibilidades, que se pueden encontrar en la documentación de la biblioteca. Por ejemplo, empleando el método `select()`, es posible buscar elementos que son *descendientes* de otro, donde el concepto de descendiente se refiere a los hijos, a los hijos de los hijos, y así sucesivamente.

Por ejemplo, otra forma de obtener el primer valor `div`, asegurando que está dentro, esto es, es descendiente del elemento `html` (lo que sucede siempre, pero sirva como ejemplo):

```
>>> print(soup.select("html div")[0].get_text())
```

Ahora vamos a ver una aplicación de *web scraping* estático sobre una página real.

Ejemplo: día y hora oficiales

Supongamos que estamos desarrollando una aplicación en la que en cierto punto es fundamental conocer la fecha y hora oficiales. Desde luego, podríamos obtener esta información directamente del ordenador, pero no queremos correr el riesgo de obtener un dato erróneo, ya sea por error del reloj interno o porque la hora y fecha hayan sido cambiadas manualmente.

Para solventar esto, la agencia estatal del Boletín Oficial del Estado pone a nuestra disposición la página:

`https://www.boe.es/sede_electronica/informacion/hora_oficial.php`

que nos da la hora oficial en la península. Para ver cómo podemos extraer la información, Introducimos esta dirección en nuestro navegador, supongamos que se trata de Google Chrome. Ahí aparecen la hora y la fecha oficiales. Para extraerla, debemos conocer la estructura de esta página en particular. Para ello, situamos el ratón sobre la hora, pulsamos el botón derecho, y del menú que aparece elegimos la opción *Inspeccionar elemento*.

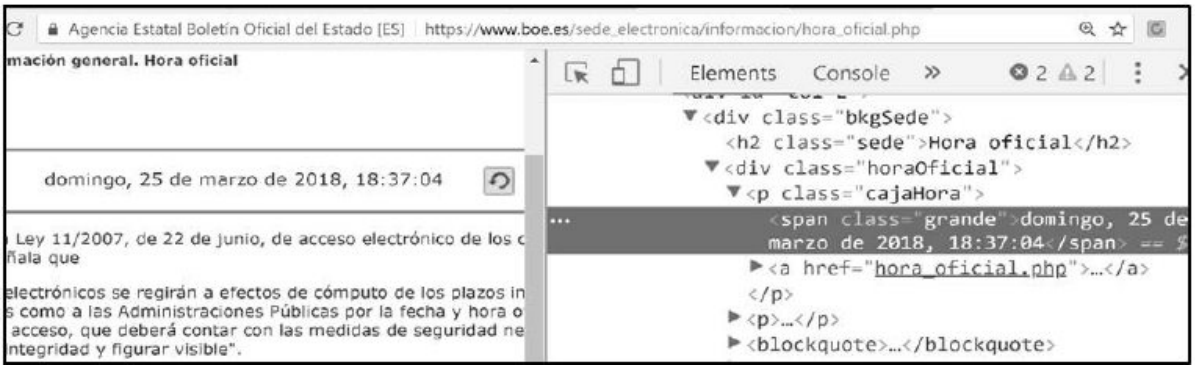


Figura 2-3. Estructura de la página del BOE con la fecha oficial.

Al hacer esto el navegador se dividirá en dos partes, tal y como muestra la figura 2-3. A la izquierda nos sigue mostrando la página web, mientras que a la derecha vemos el código HTML que define la página web, el código oculto que recibe nuestro navegador y que contiene la información que deseamos.

Vemos que la fecha y la hora corresponden al texto de un elemento `span`, con atributo `class` puesto al valor `grande`, que a su vez es hijo de un elemento `p` con atributo `class = cajaHora`. Parece que lo más sencillo es localizar el elemento `span`, por ser el más cercano al dato buscado, pero debemos recordar que este elemento lo que hace es formatear cómo se muestra un texto en la caja. Puede que se utilice el mismo formato para otras partes de la página, si no en la versión actual de la página, sí en una futura. Por ello, parece mucho más seguro emplear el elemento `p` con atributo `class = cajaHora`, cuyo nombre parece definir perfectamente el contenido.

En primer lugar, como en ejemplos anteriores, cargamos la página, mediante la biblioteca `requests`:

```
>>> import requests
>>> url = "https://www.boe.es/sede_electronica/informacion/hora_oficial.php"
>>> r = requests.get(url)
>>> print(r)
```

y a continuación buscamos el elemento `p` con `class = cajaHora` y localizamos su segundo hijo (índice 1), tras comprobar que el primer hijo es un espacio en blanco (comprobación que omitimos por brevedad):

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(r.content, "html.parser")
>>> cajaHora = soup.find("p",class_="cajaHora")
>>> print(list(cajaHora.children)[1].get_text())
```

el resultado obtenido es el día y la hora oficiales según el BOE.

Fácil, ¿verdad? Por desgracia en ocasiones la cosa se complica un poco, como veremos en el siguiente apartado.

DATOS QUE REQUIEREN INTERACCIÓN

`BeautifulSoup` es una biblioteca excelente, y nos ayudará a recuperar, de forma cómoda, datos de aquellas páginas que nos ofrecen directamente la información buscada. La idea, como hemos visto, es sencilla: en primer lugar, nos descargamos el código HTML de la página, y a continuación navegamos por su estructura, hasta localizar la información requerida.

Sin embargo, a menudo nos encontraremos con páginas que nos solicitan información que debemos completar antes de mostrarnos el resultado deseado, e decir, que exigen cierta interacción por nuestra parte. Un caso típico son las webs que nos exigen introducir usuario y palabra clave para poder entrar y acceder así a la información que deseemos. Esto no lo podemos lograr con `BeautifulSoup`.

Por ejemplo, puede que queramos saber los datos catastrales (tamaño de la finca, etc.) a partir de una dirección o de unas coordenadas. En este caso, debemos consultar la página web de la sede electrónica del catastro:

`https://www1.sedecatastro.gob.es/CYCBienInmueble/OVCBusqueda.aspx`

La página nos dará la información, pero previamente nos pedirá que introduzcamos los datos tal y como se aprecia en la figura 2-4. Para lograr esta vamos a utilizar la biblioteca *Selenium,* que no fue pensada inicialmente para hacer *web scraping*, sino para hacer *web testing*, esto es, para probar automáticamente aplicaciones web.

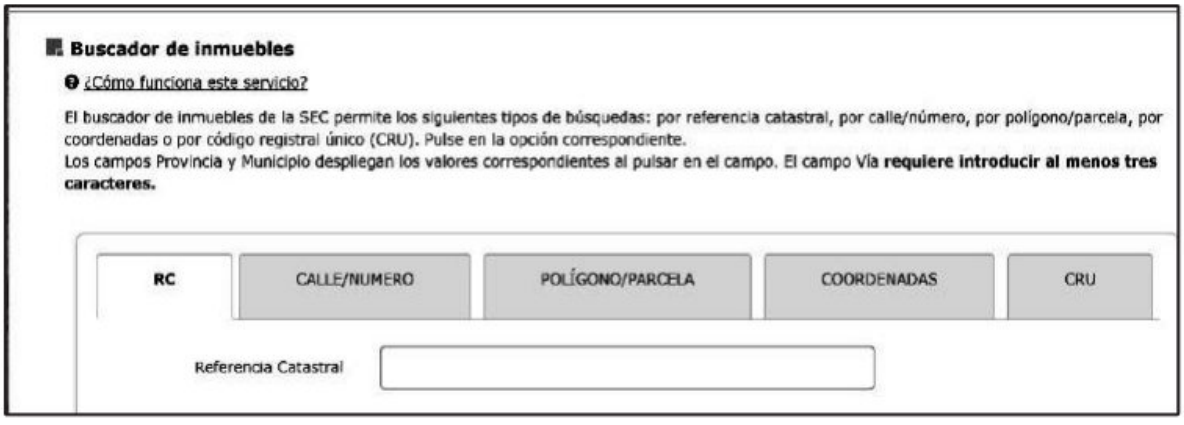


Figura 2-4. Buscador de inmuebles, sede electrónica del catastro.

La biblioteca `iniciará` una instancia de un navegador, que puede ser por ejemplo Chrome, Firefox o uno que no tenga interfaz visible, y permitirá:

- Cargar páginas.
- Hacer clic en botones.
- Introducir información en formularios.
- Recuperar información de la página cargada.

En general, `Selenium` nos permite realizar cualquiera de las acciones que realizamos manualmente sobre un navegador.

Selenium: instalación y carga de páginas

Para utilizar Selenium debemos, en primer lugar, instalar el propio paquete. La instalación se hace de la misma forma que cualquier paquete Python; el método estándar es tecleando en el terminal:

```
python -m pip install selenium
```

La segunda parte de la instalación es específica de Selenium. Para ello, debemos instalar el cliente del navegador que se desee utilizar. Los más comunes son:

- Firefox, cuyo cliente para Selenium es llamado `geckodriver`. Podemos descargarlo en:
  `https://github.com/mozilla/geckodriver/releases`
- Google Chrome: el cliente para Selenium, llamado `chromedriver`, se puede obtener en:
  `https://github.com/SeleniumHQ/selenium/wiki/ChromeDriver`

Hay más *drivers*, que podemos encontrar en la página de Selenium: Opera, Safari, Android, Edge y muchos otros.

En cualquier caso, tras descargar el fichero correspondiente debemos descomprimirlo, obteniendo el ejecutable. Aún nos queda una cosa por hacer: debemos lograr que el fichero ejecutable sea visible desde nuestra aplicación Python. Hay dos formas posibles de lograr esto.

ACCESO AL *DRIVER* A TRAVÉS DE LA VARIABLE DE ENTORNO PATH

La variable de entorno PATH es utilizada por los sistemas operativos para saber en qué lugar debe buscar ficheros ejecutables. Por tanto, una forma de hacer visible el *driver* del navegador dentro de nuestra aplicación es modificar esta variable,

haciendo que incluya la ruta hasta el ejecutable. La forma de hacer esto dependerá del sistema operativo:

- **Linux**: si nuestra consola es compatible con bash, podemos escribir:
  `export PATH = $PATH:/ruta/a/la/carpeta/del/ejecutable`

- **Windows**: buscar en el inicio "Sistema", y allí "Configuración avanzada del sistema" y "Variables de Entorno". Entre las variables, seleccionar la variable *Path*, pulsar el botón *Editar*, y en la ventana que se abre ("Editar Variables de entorno") pulsar *Nuevo*. Allí añadiremos la ruta donde está el fichero *.exe* que hemos descargado, pero sin incluir el nombre del fichero y sin la barra invertida (\) al final. Por ejemplo:

  `C:\Users\Bertoldo\Downloads\selenium`

- **Mac**: si tenemos instalado *Homebrew*, bastará con teclear
  `brew install geckodriver`
  que además lo añade al PATH automáticamente (análogo para Chrome).

En todo caso, tras modificar la variable PATH, conviene abrir un nuevo terminal para que el cambio se haga efectivo. Ahora podemos probar a cargar una página, sustituyendo el contenido de la variable `url` por la dirección web que deseemos:

```
>>> from selenium import webdriver
>>> driver = webdriver.Chrome()
>>> url    = " … "
>>> driver.get(url)
```

Si la carpeta que contiene el *driver* ha sido añadida correctamente al PATH, el efecto de este código será:

1. Abrir una instancia del navegador Chrome.
2. Cargar la página contenida en la variable `url`. En nuestro ejemplo sugerimos cargar la página:
   `https://www1.sedecatastro.gob.es/CYCBienInmueble/OVCBusqueda.asp`

ACCESO AL DRIVER INCORPORANDO LA RUTA EN EL CÓDIGO PYTHON

En ocasiones no podremos modificar el PATH, quizás por no tener permisos para hacer este tipo de cambios en el sistema. Una solución, en este caso, es incorporar la ruta al ejecutable del *driver* dentro del propio código. Para ello reemplazamos la línea `driver = webdriver.Chrome()` del ejemplo anterior por:

```
>>> import os
>>> chromedriver = "/ruta/al/exe/chromedriver.exe"
>>> os.environ["webdriver.chrome.driver"] = chromedriver
>>> driver = webdriver.Chrome(executable_path=chromedriver)
```

En este código debemos sustituir el valor de la variable `chromedriver` por la ruta al driver ejecutable. El resultado debe ser el mismo que en el caso anterior: se abrirá una nueva instancia de Chrome en la que posteriormente podremos cargar la página.

Clic en un enlace

Ya tenemos nuestra página cargada en el *driver*; podemos comprobarlo en el navegador que se ha abierto. Ahora, supongamos que deseamos conocer los datos catastrales a partir de las coordenadas (longitud y latitud). La página del catastro incluye esta opción, pero no por defecto. Por ello, antes de introducir las coordenadas debemos pulsar sobre la pestaña "COORDENADAS". Para ver el código HTML asociado, nos situamos sobre la pestaña, hacemos clic en el botón derecho del ratón y seleccionamos "inspeccionar".

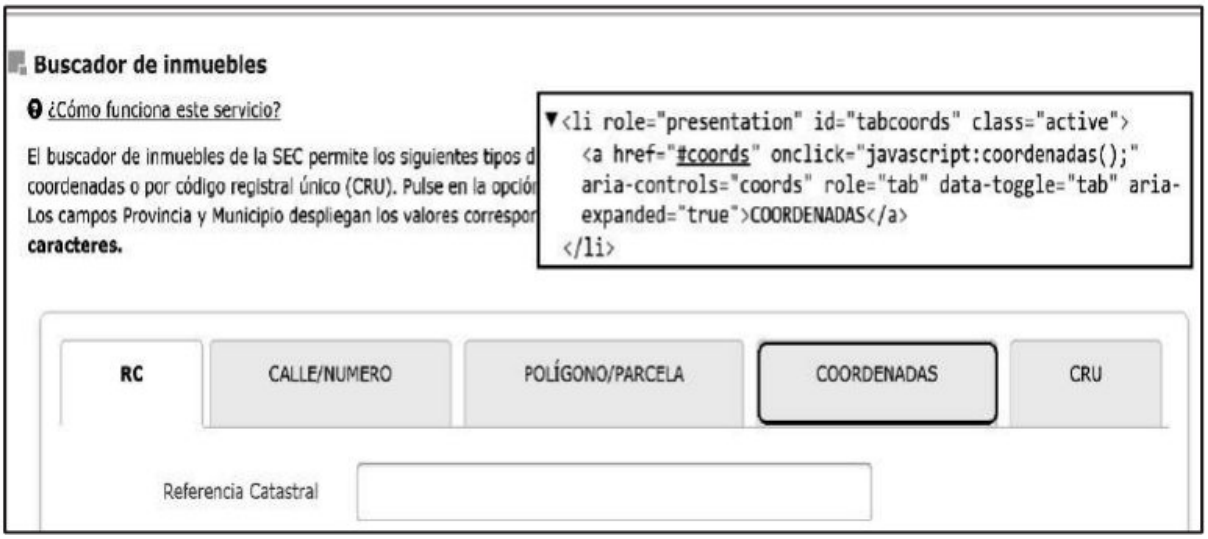


Figura 2-5. Código asociado a la pestaña COORDENADAS.

La figura 2-5 muestra el código HTML sobre la imagen de la página web (en el navegador aparecerá a la derecha, la mostramos así por simplicidad). Vemos que la palabra "COORDENADAS" aparece dentro de un elemento <a …> . En HTML estos elementos son conocidos como link*s* (enlaces). Al pulsar sobre ellos nos redirigen a otra página web, o bien a otra posición dentro de la misma página. Selenium dispone de un método que nos permite acceder a un elemento de tipo enlace a partir del texto que contiene.

```
>>> coord = driver.find_element_by_link_text("COORDENADAS")
>>> coord.click()
```

En la primera instrucción, seleccionamos el elemento de tipo link, que Selenium busca automáticamente en la página a partir del texto "COORDENADAS". En la siguiente hacemos clic sobre este elemento, con lo que la página cambiará, mostrando algo similar a la figura 2-6.

Figura 2-6. Página del catastro que permite buscar por coordenadas.

Cómo escribir texto

Ya estamos listos para introducir las coordenadas. Para ello repetimos el mismo proceso: primero nos situamos en la casilla de una de las coordenadas (por ejemplo, latitud), pulsamos el botón derecho y elegimos "inspeccionar". La figura 2-7 muestra el código asociado a este dato.

```
▼<div class="col-md-3 col-sm-3">
   <input name="ctl00$Contenido$txtLatitud" type="text" id="ctl00_Contenido_txtLatitud"
   class="form-control" onkeyup="javascript:activaBotonesCoordenadas();" oninput=
   "javascript:activaBotonesCoordenadas();" onkeypress="javascript:return
   soloDecimales(this,event);" placeholder="Ejemplo: 40.71277837">
  </div>
```

Figura 2-7. Código asociado la inspección de la caja de texto "latitud".

Vemos un elemento `div` que contiene un elemento `input`. Podemos usar el identificador (`id="ctl00_Contenido_txtLatitud"`) de este elemento para seleccionarlo. De paso hacemos lo mismo con el campo de texto de la longitud:

```
>>> lat=driver.find_element_by_id("ctl00_Contenido_txtLatitud")
>>> lon=driver.find_element_by_id("ctl00_Contenido_txtLongitud")
```

De esta forma las variables `lat` y `lon` contendrán los elementos correspondientes a las cajas de texto de la latitud y de la longitud, respectivamente.

Ahora podemos introducir la latitud y la longitud utilizando el método `send_keys()`:

```
>>> latitud  = "28.2723368"
>>> longitud = "-16.6600606"
>>> lat.send_keys(latitud)
>>> lon.send_keys(longitud)
```

Una nota final: en ocasiones la llamada a `send_keys()` puede devolver errores como:

```
WebDriverException: Message: unknown error: call function result
missing 'value'
  (Session info: chrome=65.0.3325.181)
  (Driver info: chromedriver=2.33.506120
(e3e53437346286c0bc2d2dc9aa4915ba81d9023f),platform=Windows NT
10.0.16299 x86_64)
```

Si este es el caso, tenemos un problema de compatibilidad entre nuestro navegador Chrome y el *driver*, que podremos solucionar actualizando el driver a la versión más reciente.

Pulsando botones

Aunque ya hemos introducido las coordenadas, que corresponden por cierto a la cima del Teide (Tenerife), todavía debemos pulsar el botón correspondiente para que la web del catastro haga la búsqueda y nos muestre la información deseada. Para ello repetimos, una vez más, el mismo proceso que hicimos con la pestaña "COORDENADAS": nos situamos sobre el botón "DATOS", pulsamos el botón derecho del ratón, y elegimos la opción "inspeccionar". Allí veremos los datos del botón, y en particular el atributo `id`, que cuando existe es una buena forma de identificar un elemento. Por tanto, podemos seleccionar el elemento:

```
>>> datos=driver.find_element_by_id("ctl00_Contenido_btnDatos")
```

y a continuación hacer clic:

```
>>> datos.click()
```

Tras unos instantes, la web nos mostrará los datos deseados, parte de los cuales muestra la figura 2-8.

Figura 2-8. Parte de la información mostrada para las coordenadas del ejemplo.

Es muy importante observar que tras la llamada a `datos.click()`, el contenido de la variable `driver` ha cambiado, y que ahora contendrá la nueva página web, la de los resultados. Estos "efectos laterales" pueden confundirnos a veces por lo que conviene remarcarlo: en Selenium, siempre que pulsemos sobre un enlace o un botón estaremos cambiando la página accesible desde el driver.

Si en algún momento queremos volver a la página anterior, lo más fácil es utilizar un poco de JavaScript ejecutado desde el propio driver:

```
>>> driver.execute_script("window.history.go(-1)")
```

El código indica que la página se debe sustituir por la anterior en el historial del navegador. Por cierto, ya que hemos probado este código, antes de proseguir con el capítulo conviene que hagamos:

```
>>> driver.execute_script("window.history.go(+1)")
```

para volver a la página de los resultados del catastro (figura 2-8), ya que vamos a suponer que estamos en esta página para el apartado siguiente.

Recordemos que nuestro objetivo es recopilar alguno de estos datos (por ejemplo, conocer la "clase" de terreno, o la referencia catastral). Podríamos utilizar `BeautifulSoup`, porque ya estamos en una página con los datos que deseamos, pero también podemos utilizar la propia biblioteca `Selenium`, que contiene un poderoso entorno para navegar y extraer información de páginas web.

Localizar elementos

Hasta ahora hemos empleado alguno de los métodos que tiene `Selenium` para localizar elementos, pero hay muchos más:

- `find_element_by_id()`: encuentra el primer elemento cuyo atributo `id` se le indica.
- `find_element_by_name()`: análogo, pero buscando un elemento con atributo `name`.
- `find_element_by_class_name()`: análogo, pero buscando el valor del atributo `class`.
- `find_element_by_xpath()`: búsqueda utilizando sintaxis XPath, que veremos en seguida.
- `find_element_by_link_text()`: primer elemento `<a…>text</a>` según el valor `text`.
- `find_element_by_partial_link_text()`: análogo, pero usa un prefijo del texto.
- `find_element_by_tag_name()`: primer elemento con el nombre indicado.
- `find_element_by_css_selector()`: búsqueda utilizando la sintaxis CSS.

Todos estos métodos generarán una excepción `NoSuchElementException` si no se encuentra ningún elemento que cumpla la condición requerida.

Cada uno de estos métodos tiene un equivalente que encuentra no en el primer elemento, sino en *todos* los elementos que verifiquen la condición dada. El nombre de estos métodos se obtiene a partir de los métodos ya mencionados cambiando `element` por el plural `elements`. Por ejemplo, podemos usar

```
driver.find_element_by_tag_name('div')
```

para obtener el primer elemento de tipo div, del elemento, y

```
driver.find_elements_by_tag_name('div')
```

para obtener todos los elementos de tipo `div` en una lista. La única excepción es `find_element_by_id()`, que no tiene método "plural" porque se supone que cada identificador debe aparecer una sola vez en una página web.

XPath

XPath es un lenguaje de consultas que permite obtener información a partir de documentos XML. Selenium lo utiliza para "navegar" por las páginas HTML debido a que resulta muy sencillo de aprender y a la vez ofrece gran flexibilidad. Aquí vamos a ver los elementos básicos. Una descripción en detalle se puede consultar en

`https://www.w3schools.com/xml/xpath_intro.asp`

o en el documento detallado describiendo el lenguaje, más complejo, pero interesante en caso de dudas concretas:

`https://www.w3.org/TR/xpath/`

Conviene recordar que ya hemos visto métodos que nos permiten seleccionar elementos a partir de sus atributos `id`, `class` o `name`. Sin embargo, en ocasiones no dispondremos de estos elementos, y las búsquedas serán un poco más complicadas.

XPath va a permitirnos especificar una ruta a un elemento mediante consultas, que se escribirán en Selenium entre comillas. Vamos a ver algunos de los componentes que se pueden usar en esta cadena. En los ejemplos que siguen, suponemos que estamos en la página del catastro que muestra los datos para las coordenadas 28.2723368, -16.6600606, y que se muestra en la figura 2-8.

COMPONENTE "/"

Si se usa al principio de la cadena XPath hace referencia a que vamos a referirnos al comienzo del documento (camino absoluto). Por ejemplo:

```
>>> html = driver.find_element_by_xpath("/html")
>>> print(html.text)
```

Aquí "/" indica que seleccionamos el elemento `html` que cuelga de la raíz del documento. El `print()` siguiente nos muestra una versión textual del contenido del elemento. Buena parte de la flexibilidad de XPath es que permite encadenar varios pasos. Recordemos que toda página HTML comienza con un elemento `html` que tiene dos componentes (dos hijos): `head` y `body`. Podemos extraer estos componentes de la siguiente forma:

```
>>> head = driver.find_element_by_xpath("/html/head")
>>> body = driver.find_element_by_xpath("/html/body")
```

Por ejemplo, la primera instrucción se puede leer como "ve a la raíz del documento, y busca un hijo con nombre `html`". Para este hijo, busca a su vez un elemento hijo con nombre `head`.

Recordemos que el elemento debe existir, si por ejemplo intentamos:

```
>>> otro = driver.find_element_by_xpath("/html/otro")
```

Obtendremos una excepción `NoSuchElementException`. Aquí no lo hacemos por simplicidad, pero en una aplicación real debemos utilizar estructuras `try-catch` para prevenir que el programa se "rompa" si la estructura del código HTML ha cambiado, o simplemente si introducimos un camino que no existe por error.

Un aspecto importante que debemos tener en cuenta es que cualquier búsqueda en Selenium devuelve un elemento de tipo `WebElement`

, que es un *puntero* al elemento(s) seleccionado(s). No debemos pues pensar que la variable obtenida "contiene" el elemento, sino más bien que "señala" al elemento. Esto explica que el siguiente código, aunque ciertamente extraño y poco recomendable, sea legal:

```
>>> html2 = body.find_element_by_xpath("/html")
```

Utilizamos `body` y no `driver` como punto de partida, cosa que haremos pronto para construir caminos paso a paso. Pero lo interesante es que forzamos acceder de nuevo a la raíz y tomar el elemento `html`. Esto no podría hacerse si `body` fuera realmente el cuerpo del código HTML; no podríamos acceder desde dentro de él a un elemento exterior. Pero funciona porque Selenium simplemente va a la raíz del documento al que apunta `body`, que es el documento contenido en `driver`, y devuelve el elemento `html`.

En particular esta idea de los "señaladores" nos lleva a entender que si ahora cambiamos la página contenida en el `driver`, la variable `body` dejará de tener un contenido válido, porque "apuntará" a un código que ya no existe. Por ejemplo, el siguiente código (que no recomendamos ejecutar ahora para no cambiar la página de referencia) provocará una excepción:

```
>>> driver.execute_script("window.history.go(-1)")
>>> print(body.text)
```


COMPONENTE "*"

Cuando no nos importa el nombre del elemento concreto, podemos utilizar "*", que representa a cualquier hijo a partir del punto en el que se encuentre la ruta. Por ejemplo, para ver los nombres de todos los elementos que son hijos de `body` podemos escribir el siguiente código:

```
hijos = driver.find_elements_by_xpath("/html/body/*")
for element in hijos:
      print(element.tag_name)
```

Otra novedad de este ejemplo es la utilización de `find_elements_by_xpath()` en lugar de `find_element_by_xpath()` para indicar que en lugar de un elemento queremos que se devuelvan *todos* los elementos que cumplen la condición. El resultado es una lista de objetos de tipo `WebElement`, almacenada en la variable `hijos`. Para recorrer la lista utilizamos un bucle que va mostrando el nombre de cada elemento hijo (atributo `tag_name` de `WebElement`). En nuestro caso mostrará:

```
form
script
a
iframe
```

El primer elemento es un formulario, el segundo un script con código JavaScript, el tercero un enlace (representado en HTML por `a`), y el último un `iframe`, que es una estructura empleada en HTML para incrustar un documento dentro de otro.

El elemento "*" puede ser seguido por otros. Por ejemplo, supongamos que queremos saber cuántos elementos de tipo `div` son "nietos" de `body`.

```
>>> divs = driver.find_elements_by_xpath("/html/body/*/div")
>>> print(len(divs))
```

que nos mostrará el valor 4.

COMPONENTE "."

El punto sirve para indicar que el camino sigue desde la posición actual. Resulta muy útil cuando se quiere seguir la navegación desde un "señalador". Por ejemplo, otra forma de obtener los "nietos" de tipo `div` del elemento `body`, es partir del propio `body`:

```
>>> divs = body.find_elements_by_xpath("./*/div")
>>> print(len(divs))
```

COMPONENTE "//"

Las dos barras consecutivas se usan para saltar varios niveles. Por ejemplo, supongamos que queremos saber cuántos valores `div` son descendientes de `body`, es decir, cuántos `div` están contenidos dentro de `body` a cualquier nivel de profundidad. Podemos escribir:

```
>>> divs = driver.find_elements_by_xpath("/html/body//div")
>>> print(len(divs))
```

que devolverá el valor `108` en nuestro ejemplo. También podemos preguntar por todos los elementos `label` del documento:

```
>>> labels = driver.find_elements_by_xpath("//label")
>>> print(len(labels))
```

que muestra el valor 6.

Por ejemplo, supongamos que queremos encontrar la referencia catastral incluida en la página de nuestro ejemplo. Como siempre, nos situamos sobre ella y pulsamos el botón derecho del ratón, eligiendo *inspeccionar*. El entorno código HTML que precede a la referencia buscada es de la forma:

```
…
<div id="ctl00_Contenido_tblInmueble"
     class="form-horizontal" name="tblInmueble">
  <div class="form-group">
      <span class="col-md-4 control-label ">
            Referencia catastral
      </span>
       <div class="col-md-8 ">
         <span class="control-label black">
           <label class="control-label black text-left">
                          38026A035000010000EI 
…
```

La referencia (el valor `38026A035000010000EI`), se encuentra dentro de un elemento de tipo `label`, pero como hemos visto el documento contiene 6 valores de este tipo. Lo usual en estos casos es buscar un elemento con identificador cercano, en este caso el `div` de la primera línea del código que mostramos (`id="ctl00_Contenido_tblInmueble"`), y utilizarlo de punto de partida para localizar el valor buscado:

```
>>> id = "ctl00_Contenido_tblInmueble"
>>> div = driver.find_element_by_id(id)
>>> label = div.find_element_by_xpath("//label")
>>> print(label.text)
```

Primero hacemos que la variable `div` apunte al elemento `div` con el identificador indicado. A continuación, buscamos la primera etiqueta (elemento `label`) descendiente de este elemento `div`. Finalmente mostramos el texto, que es el valor deseado:

```
38026A035000010000EI
```

En el código hemos utilizado las versiones "singulares", es decir, `find_element()` en lugar de `find_elements()`. En el primer caso, `find_element_by_id()`, porque estamos buscando un `id`, que se supone siempre único. Y en el segundo caso, `find_element_by_xpath()`, porque estamos buscando la primera etiqueta descendiente del elemento `div`.

La ventaja del uso de "`//`" es que logramos cierta independencia del resto de la estructura, es decir la consulta seguiría funcionando siempre que el elemento con `id = "ctl00_Contenido_tblInmueble"` siga existiendo, y su primera etiqueta hija contenga el valor buscado. El único inconveniente es que aún utilizamos la posición de la etiqueta, lo que indica que, si se añadiera en el futuro otra etiqueta más arriba, sería la recién llegada la seleccionada, devolviendo información errónea.

FILTROS `[ … ]`

Los filtros son una herramienta poderosa de XPath. Permiten indicar condiciones adicionales que deben cumplir los elementos seleccionados. La forma más simple de filtrado es simplemente indicar la posición de un elemento particular. Por ejemplo, supongamos que deseamos saber qué tipo de finca es la que corresponde a esta referencia catastral. Si miramos de nuevo la figura 2-8, veremos que este valor viene en tercer lugar y corresponde a la "clase" de terreno, que en este caso es `Rústico`. Podemos entonces obtener directamente la tercera etiqueta:

```
>>> e = driver.find_elements_by_xpath(
>>>       "(//label)[position()=3]")
>>> print(e[0].text)
```

que mostrará por pantalla el texto `Rústico` tal y como deseábamos. El uso de los paréntesis alrededor de `//label` es importante, porque el operador `[]` tiene más prioridad que "`//`", y de otra forma obtendríamos un resultado erróneo.

También observamos que, aunque hemos seleccionado un solo elemento, seguimos recibiendo una lista de `WebElement`, aunque sea de longitud unitaria. Por eso en el `print()` tenemos que usar la notación `e[0]`.

Esta aplicación de los filtros es tan usual que XPath nos permite eliminar la llamada a `position()`, dando lugar a una notación típica de los *arrays* de programación:

```
>>> e = driver.find_elements_by_xpath("(//label)[3]")
>>> print(e[0].text)
```

Es importante que señalar que en XPath *el primer elemento tiene índice 1, y no 0 como en Python*. Por eso, si queremos lograr el mismo efecto usando Python, podemos devolver todas las etiquetas y seleccionar la de índice 2, en lugar de 3 como era el caso en XPath:

```
>>> etiqs = driver.find_elements_by_xpath("//label")
>>> print(etiqs[2].text)
```

El código lógicamente mostrará el mismo valor, `Rústico`. XPath también nos permite acceder al último elemento, usando la función `last()`. En nuestro ejemplo podemos por ejemplo obtener el área de la finca, que es la última etiqueta:

```
>>> ulti = driver.find_elements_by_xpath(
       "(//label)[last()]")
>>> print(ulti[0].text)
```

que nos mostrará "`68.167.075 m2`". Análogamente, podemos usar `last()-1` para referirnos al penúltimo elemento.

Otro de los usos más habituales de los filtros es quedarse con los elementos con valores concretos para un atributo. Esto se logra antecediendo el nombre del atributo por "@":

```
>>> xpath =
       "//label[@class='control-label black text-left']"
>>> etiqs = driver.find_elements_by_xpath(xpath)
>>> print(etiqs[0].text)
```

La consulta XPath es un poco más compleja y merece ser examinada en detalle:

- En primer lugar, "`//label`" indica que buscamos elementos `label` a cualquier nivel dentro del documento.

- A continuación, el filtro `[@class='control-label black text-left']` pide que, de todas las etiquetas recolectadas, la consulta seleccione solo las que incluyan un atributo `class` que tome el valor `'control-label black text-left'`.

Como el valor de la referencia catastral es el único del documento con una etiqueta de esta clase, el fragmento de código muestra justo este valor, `38026A035000010000EI`.

Por último, veamos una forma más compleja de obtener este mismo valor, que puede presentar ciertas ventajas. Supongamos que todo lo que podemos asegurar es que:

- Sabemos que la referencia catastral será el texto de un elemento `label`.
- También sabemos que este elemento `label` será *hermano* de otro elemento de tipo `span` que tendrá como texto "`Referencia catastral`".

Estas suposiciones se obtienen de la estructura del documento, y no utilizan criterios como la posición ni los atributos particulares del elemento `label`. En XPath lo podemos codificar así:

```
>>> xpath =
>>>    "//*[./span/text()='Referencia catastral']//label"
>>> etiq = driver.find_element_by_xpath(xpath)
>>> print(etiq.text)
```

El resultado es, una vez más, el valor catastral. Vamos a explicar en detalle la consulta:

- Buscamos un elemento *P* de cualquier tipo y en cualquier lugar (`//*`)tal que:
  - Tenga un hijo (`./`), de tipo `span` que incluya como texto "`Referencia catastral`".
- Una vez encontrado este elemento *P*, nos quedamos con su hijo de tipo `label`.

Una nota final: cuando hayamos terminado de utilizar Selenium conviene escribir `driver.close()` para liberar recursos, especialmente si el código está dentro de un bucle que se va a repetir varias veces.

Navegadores *headless*

Trabajar desde nuestra aplicación en Selenium con un navegador abierto como en los ejemplos anteriores tiene la ventaja de que podremos ver cómo funciona la aplicación: cómo se escribe el texto, se hace clic, etc. Sin embargo, en una aplicación real normalmente no deseamos que se abra una nueva instancia del navegador; resultaría muy sorprendente para el usuario ver que de repente se le abre Chrome y empieza a cargar páginas, a rellenar datos, etc. Incluso puede suceder que lo cierre y de esa forma detenga la ejecución del programa. Si queremos evitar esto, emplearemos un navegador *headless*, nombre con el que se denominan a los navegadores que no tienen interfaz gráfica. En el caso de Chrome esto podemos lograrlo simplemente añadiendo opciones adecuadas a la hora de crear el driver:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.chrome.options import Options
>>> chrome_options = Options()
>>> chrome_options.add_argument('--headless')
>>> chrome_options.add_argument(
       '--window-size=1920x1080')
>>> driver = webdriver.Chrome(
       chrome_options=chrome_options)
```

Es especialmente llamativo el hecho de que además de indicar como argumento que el navegador debe arrancar en modo *headless*, es decir, sin interfaz gráfica, digamos también el tamaño de la ventana que debe ocupar. Es conveniente hacer esto porque, aunque no veamos el navegador, internamente sí se crea y en algunos casos puede dar error si no dispone de espacio suficiente para crear todos los componentes. En el navegador estándar esto no sucede porque se adapta automáticamente a la pantalla. En un navegador *headless*, en cambio, tenemos que indicar nosotros un tamaño suficientemente grande.

CONCLUSIONES

Aunque Python nos ofrece varias librerías como Selenium y BeautifulSoup que nos facilitan la tarea, obtener información de la web mediante *web scraping* es una tarea que requiere conocer muy bien la estructura de la página que contiene la información y que lleva tiempo de programación. Merecerá la pena sobre todo si es una tarea que vamos a realizar de forma reiterada.

En el siguiente capítulo, vamos a ver otra forma de acceder a los datos, a través de servicios proporcionados para tal fin.

REFERENCIAS

- Documentación de la biblioteca BeautifulSoup (accedido el 30/06/2018). https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- Documentación de la biblioteca Selenium (accedido el 30/06/2018). https://www.seleniumhq.org/docs/
- Ryan Mitchell. *Web Scraping with Python: Collecting More Data from the Modern Web*. 2ª edición. O'Reilly Media, 2018.
- Richard Lawson. *Web Scraping with Python (Community Experience Distilled)*. Packt, 2015.
- Mark Collin. *Mastering Selenium WebDriver*. Packt, 2015.
- Vineeth G. Nair. *Getting Started with Beautiful Soup*. Pack. 201

3 RECOLECCIÓN MEDIANTE APIS

INTRODUCCIÓN

Como hemos visto en capítulos anteriores, en algunas páginas es posible descargar directamente un fichero con los datos que buscamos. Sin embargo, en páginas en las que los datos son más ricos o cambian más habitualmente no tiene sentido seguir este modelo, pues sería necesario actualizar cada día ficheros de gran tamaño de los que el usuario solo necesita una pequeña parte o, sencillamente, porque los datos son demasiado grandes.

Imaginemos, por ejemplo, qué tipo de ficheros deberíamos proporcionar para almacenar los tweets generados en un solo día: un solo fichero tendría un tamaño descomunal. En cambio, si repartiésemos los tweets por *hashtags* (como veremos, cadenas de texto que empiezan por #) generaríamos un número de ficheros demasiado elevado y con muchas repeticiones. Si en lugar de por hastags, dividiéramos los ficheros por localizaciones geográficas perderíamos la información de aquellos que tienen desactivada esta opción... Es necesario, por tanto, otro mecanismo mediante el cual podamos recolectar la información de interés de manera sencilla y eficiente.

En este capítulo veremos cómo usar diversas interfaces de programación de aplicaciones (API por sus siglas en inglés, *Application Programming Interface*). Una API es una biblioteca, es decir, un conjunto de funciones que ofrece una cierta aplicación para ser accedida por otra. En nuestro caso estamos interesados en las

APIs desde Python, de las que presentaremos dos ejemplos: el API de Twitter y el API REST de OMDB.

API TWITTER

En esta sección vamos a ver cómo acceder a Twitter, descargando los mensajes que nos interesen, ya sea consultando los mensajes ya existentes, o "escuchando" según se emiten (lo que se conoce como acceso en *streaming*).

Partimos del supuesto de que el lector tiene un conocimiento a nivel de usuario de lo que es Twitter y que, por tanto, conoce términos como *hashtag*, *trending topic* o *retweet*. En caso contrario, recomendamos consultar la ayuda en español de Twitter, disponible en las referencias.

El API que usaremos para acceder a los datos de Twitter desde Python en modo *streaming* es *Tweepy*. Para poder utilizar esta librería y acceder a sus datos de forma gratuita, Twitter requiere que nos demos de alta como desarrolladores. En particular, nuestra aplicación debería contener unos *tokens* o claves personales. Estos valores van asociados a nuestra cuenta de Twitter (o a la cuenta que creemos para la ocasión) y a la aplicación concreta, que debe ser dada de alta.

Vamos a ver, en primer lugar, qué pasos debemos seguir en nuestra cuenta de Twitter para obtener estos *tokens*. Después, veremos qué formato tienen los datos proporcionados por el API y, por último, cómo descargarlos. Durante todo el capítulo consideraremos importada la correspondiente biblioteca como:

```
>>> import tweepy
```

Acceso a Twitter como desarrollador

Supongamos que ya disponemos de una cuenta en Twitter, en nuestro caso la cuenta "Libro Python", que queremos usar para descargar tweets. Es importante percatarse que la cuenta que usemos debe haber incluido entre sus datos el teléfono del usuario; en caso contrario nos pedirán completarlos durante el proceso.

En primer lugar, accederemos a `https://apps.twitter.com`, donde indicaremos que queremos crear una nueva aplicación pulsando el botón "Create New App". Al pulsar el botón accedemos a una página en la que tenemos que dar los detalles de nuestra aplicación, como se muestra en la figura 3-1. En esta vista debemos introducir el nombre de nuestra aplicación, una descripción y una dirección web en la cual esté disponible la información sobre los datos usados y los resultados obtenidos.

Una vez introducidos estos datos leeremos el acuerdo de desarrollador y, en caso de aceptarlo, marcaremos la casilla correspondiente y pulsaremos el botón "Create your Twitter application".

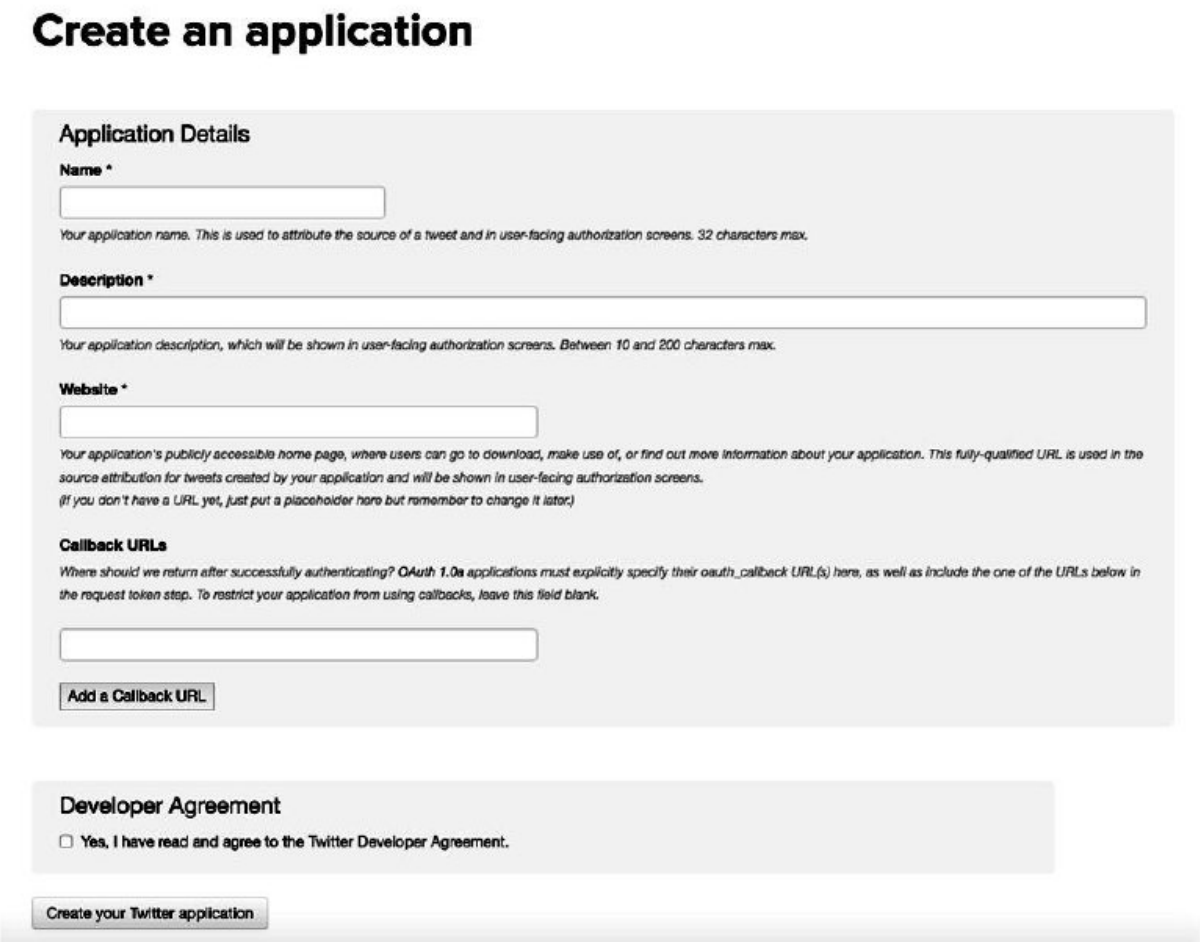


Figura 3-1. Registro de aplicación.

Tras pulsar este botón llegaremos a una página con 4 pestañas en la parte superior, tal y como se muestra en la figura 3-2. En esta página encontramos la información que hemos dado sobre nuestra aplicación, así como algunas direcciones para acceder a distintos *tokens* (claves de acceso que permiten verificar que somos quienes decimos ser).

Para conseguir todos los *tokens* necesarios para descargar tweets iremos a la pestaña "Keys and Access Tokens" y pulsaremos el botón "Create my access token" (figura 3-3). Esto hará que se generen las claves de acceso que, junto a las claves de consumidor que se han generado al crear la aplicación, nos permitirá completar las siguientes cuatro claves:

```
>>> CONSUMER_TOKEN = "..."
>>> CONSUMER_SECRET = "..."
>>> ACCESS_TOKEN = "..."
>>> ACCESS_TOKEN_SECRET = "..."
```

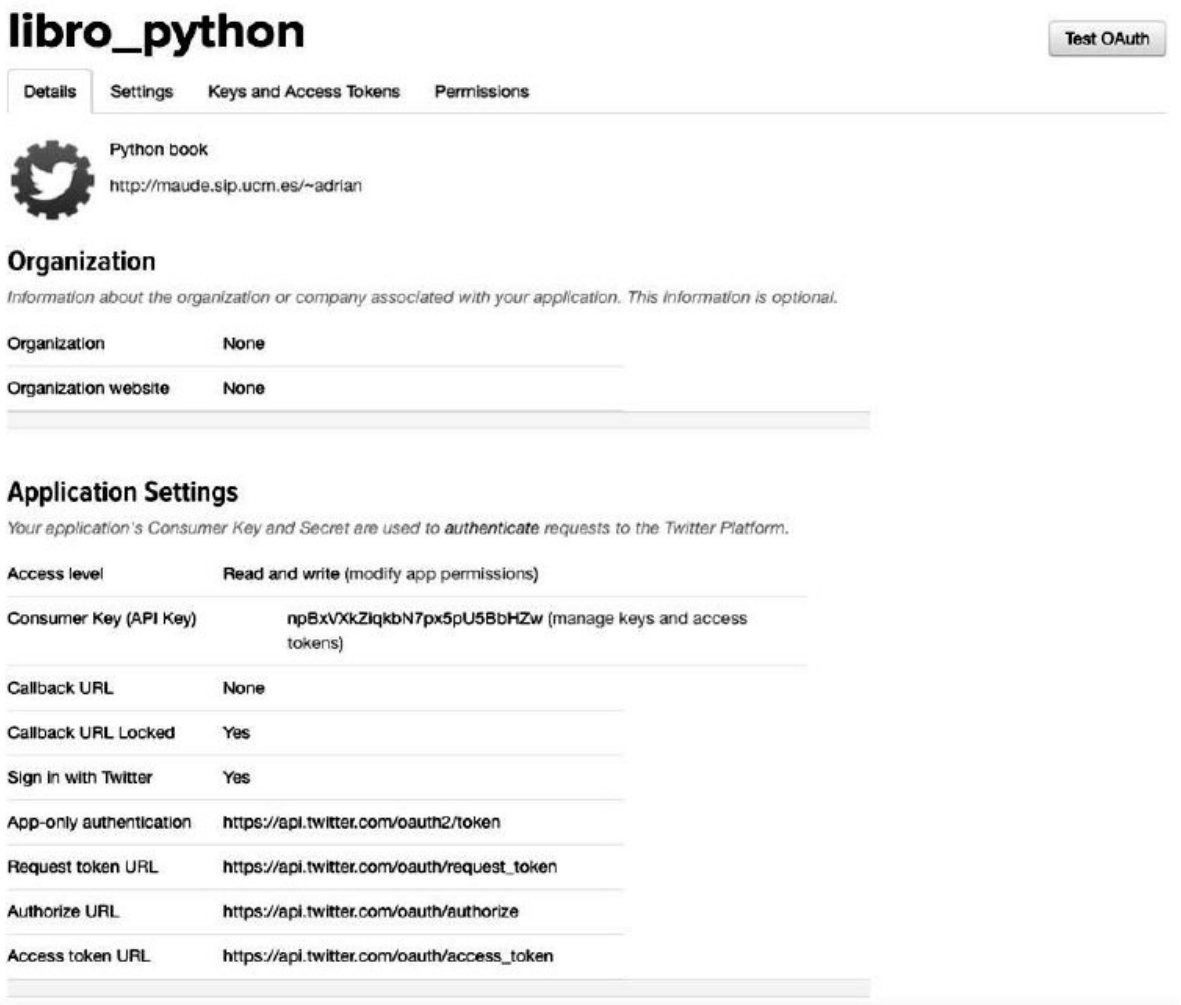


Figura 3-2. Detalles de la creación de una nueva aplicación Twitter.

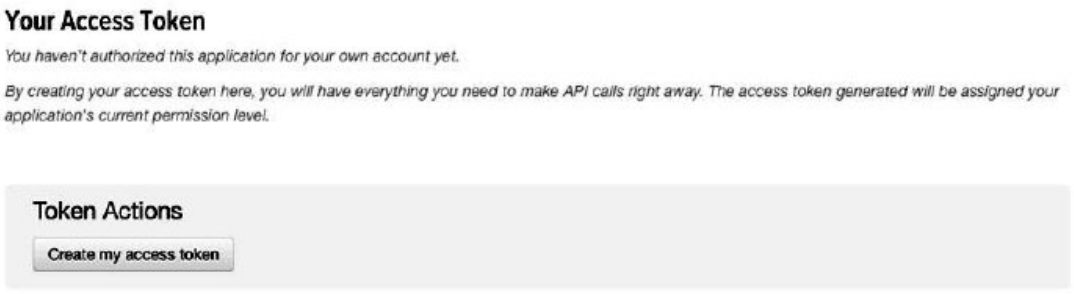


Figura 3-3. Generación de tokens de acceso.

En este momento estamos en disposición de crear una autentificación que nos sirva para usar al API. Dicha autentificación es un objeto de la clase OAuthHandler, que se crea en Python como:

```
>>> auth = tweepy.OAuthHandler(CONSUMER_TOKEN, CONSUMER_SECRET)
```

Una vez creado este objeto, introduciremos nuestros *tokens* de acceso con la función `set_access_token`:

```
>>> auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
```

Por último, obtendremos un objeto de clase `API` que servirá para realizar consultas:

```
>>> api = tweepy.API(auth)
```

En el siguiente apartado estudiaremos la estructura de los tweets que vamos descargar. Posteriormente, mostraremos dos maneras de realizar esta descarga.

Estructura de un tweet

Los tweets que descargamos con Tweepy son objetos JSON con la siguiente estructura:

- `id` e `id_str`, el identificador único del tweet tanto en formato numérico como de *string*.
- `text`, un `string` que corresponde al texto del tweet, esto es el mensaje emitido por el usuario.
- `source`, un *string* que indica el origen del tweet. En este caso el atributo indicará bien la aplicación de móvil desde la que se escribió un tweet (por ejemplo, `"Twitter for Mac"` si escribimos desde un *iPhone* o `"web"` si se escribió desde un ordenador).
- `truncated`, un booleano que indica si el tweet se ha truncado porque, a consecuencia de un retweet, se excede del límite de caracteres permitido por Twitter.
- `in_reply_to_status_id`, un entero que indica el identificador del tweet al que se está contestando, si es el caso. Análogamente, `in_reply_to_status_id_str` indica el identificador de tipo *string*. Si el tweet no es una contestación a otro, su valor será `null` en ambos casos.
- `in_reply_to_user_id`, un entero que indica el identificador del usuario que escribió el tweet del que el tweet descargado es respuesta. De la misma manera, `in_reply_to_user_id_str` hace referencia al

identificador de tipo *string*. Por último, `in_reply_to_screen_name` indica el nombre/apodo que el usuario que escribió el tweet original muestra por pantalla. Si el tweet no es una contestación, todos estos valores toman el valor `null`.

- `user`, un objeto JSON con la siguiente estructura:
    - `id`, un entero que representa el identificador del usuario. La versión de tipo *string* se almacena en `id_str`.
    - `name`, un *string* que muestra el nombre tal y como lo ha definido el usuario.
    - `screen_name`, un *string* que muestra el apodo mostrado por el usuario.
    - `location`, localización del usuario, definida por este. No tiene por qué ser una localización real, ni se requiere un formato específico (por ejemplo, puede ser `"mi casa"`). Su valor puede ser `null`.
    - `url`, *string* con una dirección web dada por el usuario en su perfil. Puede tomar el valor `null`.
    - `description`, *string* con la descripción dada por el usuario en su perfil.
    - `derived`, un objeto que contiene los metadatos de localización del usuario. Consiste en un solo objeto `locations`, que a su vez es un array de objetos. En `locations` encontramos información sobre la ciudad, el estado, las coordenadas, etc., del usuario; para no agobiar con un excesivo nivel de detalle, recomendamos al lector interesado consultar la documentación en las referencias.
    - `protected`, un booleano que indica si el usuario tiene sus tweets protegidos (es decir, solo sus seguidores pueden verlos).
    - `verified`, un booleano que indica si la cuenta está verificada.
    - `followers_count`, un entero que indica el número de seguidores del usuario.
    - `friends_count`, un entero que indica el número de amigos del usuario.
    - `listed_count`, un entero que indica de cuántas listas públicas es miembro el usuario.
    - `favourites_count`, un entero que indica cuántos tweets ha señalado como favoritos el usuario.
    - `statuses_count`, entero que indica el número de tweets (incluyendo retweets) que ha creado el usuario.
    - `created_at`, `string` que indica, en formato UTC, cuándo se creó la cuenta.
    - `utc_offset` y `time_zone`, dos elementos sobre zona horaria que en la actualidad están fijados a `null`.

    - geo_enabled, booleano que indica si el usuario tiene activada la localización geográfica.
    - lang, *string* que indica el lenguaje seleccionado por el usuario.
    - contributors_enabled, booleano que indica si el usuario permite contribuciones de co-autores con otra cuenta.
    - profile_* y default_profile_*. Hay un conjunto de atributos que empiezan por estos prefijos que almacenan información sobre cómo se muestra el perfil, incluyendo la imagen elegida por el usuario, el fondo, el color de los links, etc.
    - withheld_in_countries, *string* que indica en qué países ha sido censurado.
    - whitheld_scope, *string* que solo puede tomar los valores 'user' y 'status'. Indica si el objeto de la censura es el usuario al completo o solo uno de sus tweets.
- coordinates, objeto GeoJSON (disponible en las referencias) que indica dónde se publicó el tweet. Toma el valor null cuando el usuario no tiene activada la localización geográfica.
- place, objeto JSON que indica si el tweet está asociado a un lugar, aunque no necesariamente debe haber sido publicado allí. Puede tomar el valor null.
- quoted_status_id, entero que solo aparece si el tweet es contestación a otro e indica el identificador del tweet al que se responde. De la misma manera, quoted_status_id_str hace referencia al identificador de tipo *string*.
- is_quote_status, booleano que indica si el tweet es una cita.
- quoted_status, objeto JSON correspondiente al tweet citado. Este campo solo aparece cuando el tweet es una cita.
- retweeted_status, objeto JSON correspondiente al tweet que se ha retweeteado. Este campo solo aparece cuando el tweet es un retweet.
- quote_count, entero que indica el número de veces que el tweet ha sido citado.
- reply_count, entero que indica las respuestas que ha recibido el tweet.
- retweet_count, entero que indica el número de veces que el tweet ha sido retweeteado.
- favorite_count, entero que indica el número de veces que el tweet ha sido marcado como favorito.
- entities, objeto JSON con información sobre el texto del tweet. A continuación listamos los distintos elementos del objeto. Muchos de ellos son a su vez complejos, por lo que referimos al lector interesado a las referencias para más información:

    - `hashtags`, array de objetos JSON con información sobre los *hashtags*. Cada uno de estos objetos cuenta con el elemento `text`, que contiene el texto del *hashtag* (sin #) y el elemento `indices`, una lista de dos enteros con las posiciones en el texto en donde aparece el *hashtag*.
    - `media`, array de objetos JSON de tipo `Media` con la información multimedia que aparece en el tweet. Esta información debe haber sido compartida a través de un enlace.
    - `urls`, array de objetos `URL` con la información sobre las URLs que aparecen en el tweet.
    - `user_mentions`, array de objetos `UserMention` con información sobre los usuarios mencionados en el tweet.
    - `symbols`, array de objetos `Symbol` con información sobre los símbolos (por ejemplo, emoticonos) que aparecen en el tweet. Como en el caso de los hashtags, estos elementos se componen de un elemento `indices` con la posición del símbolo y un elemento `text` con el texto.
    - `polls`, array de objetos `Poll` con información sobre las encuestas realizadas en el tweet.
- `extended_entities`, objeto de tipo `entities` que solo contendrá, a su vez, el objeto `media`. Este elemento se usa para indicar el contenido multimedia compartido a través de la interfaz de Twitter de manera nativa, en lugar de compartirlo mediante un enlace con contenido externo.
- `favorited`, booleano que indica si el usuario que está realizando la búsqueda marcó el tweet como favorito.
- `retweeted`, booleano que indica si el usuario que está realizando la búsqueda retweeteó el tweet.
- `possibly_sensitive`, booleano que puede tomar el valor `null`. Indica si el tweet contiene algún enlace potencialmente peligroso.
- `filter_level`, un *string* que puede tomar los valores `"none"`, `"low"` y `"medium"` (está previsto el valor `"high"`, pero todavía no está disponible). Este valor está destinado a las aplicaciones que escuchan en *streaming* y necesitan hacer una selección de los datos recogidos. Así, cuanto mayor sea el nivel del filtro de mayor interés será el tweet.
- `lang`, *string* que indica el idioma inferido para el tweet o `"und"` si no se ha detectado ninguno.
- `matching_rules`, array de objetos `Rule` con información sobre búsquedas.

Como se puede ver, aunque un tweet puede parecer una cosa simple almacena una gran cantidad de información que se puede utilizar para realizar diferentes análisis. Además, los campos descritos son los utilizados en el momento de escribir este libro, pero en cualquier momento Twitter puede eliminar o añadir nuevos campos. Por ello recomendamos revisar periódicamente la documentación para conocer los cambios que hayan podido producirse.

Descargando tweets

En esta sección veremos que es posible descargar tweets de dos maneras: buscando tweets ya almacenados, o escuchando continuamente (conocido como *streaming* en inglés) lo que se publica en Twitter y guardando aquellos que cumplan ciertos criterios. Es interesante observar que en general no obtendremos todos los tweets, ya que hay limitaciones de pago en la cantidad de información a la que podemos acceder, aunque en general se puede acceder a un número mucho mayor de tweets con el método streaming.

BÚSQUEDA PUNTUAL DE TWEETS

En primer lugar, veremos cómo buscar tweets en un momento dado. Para ello usaremos la función `search` del API. Esta función admite los siguientes argumentos:

- `q`, un *string* que indica la búsqueda a realizar. Este parámetro es el único obligatorio.
- `lang`, cadena que indica el idioma de los tweets a descargar.
- `locale`, cadena que indica el idioma en el que se ha realizado la consulta.
- `rpp`, entero que indica el número de tweets por página, con un máximo de 100.
- `page`, entero que indica la página que queremos descargar, siendo 1 la primera. Por ejemplo, si realizamos una búsqueda con `rpp=100` y `page=3` obtendremos los resultados desde el 201 hasta el 300. La cantidad de elementos que se pueden buscar en total es 1500.
- `since_id`, entero que indica el identificador del tweet más antiguo que se puede devolver en la búsqueda.
- `geocode`, un *string* que indica el área donde deben haber sido publicados los tweets para ser devueltos en la búsqueda. Esta cadena es de la forma `"latitud,longitud,radio"`, donde el radio debe terminar en `"mi"` o `"km"` para indicar si está definido en millas o kilómetros, respectivamente. Si queremos realizar una búsqueda por área podemos usar `q="*"` para indicar que aceptamos todos los tweets publicados en esa área. Es

importante observar que el comodín solo se puede usar cuando damos valor a geocode, en otro caso la búsqueda no será válida.

- `show_user`, booleano que indica si se debe concatenar el nombre del autor del tweet seguido de ":" al principio del texto. Este mecanismo es útil cuando se descarta el nombre del autor; por defecto su valor es `False`.

Esta función permite descargar un máximo de 100 tweets, por lo que si queremos descargar más deberemos combinar los atributos `rpp` y `page` para avanzar la búsqueda. El resultado de aplicar `search` es una lista de objetos de clase `Status`. Los objetos de esta clase tienen un atributo `_json` que contiene el documento JSON con los campos explicados en la sección anterior. Así, el siguiente código muestra cómo descargar de manera sencilla tweets que mencionen `"python"` y obtener el correspondiente JSON, que puede ser entonces almacenado con las funciones que vimos en el primer capítulo:

```
>>> lista_tweets = api.search(q="python")
>>> lista_json = []
>>> for tweet in lista_tweets:
>>>     lista_json.append(tweet._json)
```

Es interesante apuntar que el API nos permite realizar muchas de las acciones disponibles desde la interfaz web, desde modificar nuestros mensajes a buscar nuestros amigos. Consideramos que estas funciones no están directamente relacionadas con la temática del libro, pero el lector tiene disponible más información en las referencias.

BÚSQUEDA DE TWEETS EN *STREAMING*

La búsqueda que hemos presentado en la sección anterior nos permite obtener un máximo de 1500 resultados. Difícilmente podríamos considerar un conjunto de datos de este tamaño como *big data*, por lo que es necesario buscar una manera de descargar más datos. La solución la encontramos en tener un programa continuamente "escuchando" en Twitter y descargando en tiempo real aquellos tweets que encajan en nuestros criterios de búsqueda. Para ello, en primer lugar es necesario crear una clase que herede de `tweepy.StreamListener`. Esta clase dispone de varias funciones que podemos redefinir para adecuar al comportamiento deseado:

- `on_status`, que es invocada cuando un tweet de las características buscadas es encontrado.
- `on_error`, ejecutada cuando ocurre un error.

En nuestro caso queremos que los tweets encontrados se almacenen en un fichero, por lo que pasaremos una ruta a la función constructora de la clase, que abrirá el fichero correspondiente y lo guardará como un atributo. La función `on_status` escribirá el elemento JSON almacenado en el objeto `Status` leído junto a un salto de línea, para poder leer los tweets posteriormente de manera sencilla. Por último, cuando suceda un error simplemente cerraremos el fichero:

```
class MyStreamListener(tweepy.StreamListener):

    def __init__(self, api, ruta):
        super().__init__(api)
        self.fich = open(ruta, 'a')

    def on_status(self, status):
        self.fich.write(json.dumps(status._json) + "\n")

    def on_error(self, status_code):
        self.fich.close()
```

A continuación, definiremos una ruta para nuestro fichero, crearemos un objeto de la clase anterior y la usaremos para crear un objeto de la clase `Stream`, encargado de realizar búsquedas usando las credenciales que creamos anteriormente y el objeto recién creado:

```
ruta_datos = "./datos_twitter.txt"
myStreamListener = MyStreamListener(api, ruta_datos)
flujo = tweepy.Stream(auth = auth, listener=myStreamListener)
```

Los objetos de la clase `Stream` proporcionan la función `filter`, encargada de decidir qué tweets serán elegidos por la función `on_status`. Esta función admite los siguientes argumentos:

- `follow`, una lista de identificadores de usuarios de los que queremos descargar información.
- `track`, una lista de palabras clave a buscar.
- `locations`, una lista de localizaciones de las que queremos descargar información.
- `delimited`, número de bytes máximo que debe tener el tweet.
- `stall_warnings`, indica si se deben enviar mensajes cuando la conexión comienza a fallar. Toma los valores `'true'` y `'false'`.

La siguiente llamada almacena en el fichero que hemos especificado arriba todos los tweets que mencionen la palabra `python`:

```
flujo.filter(track=['python'])
```

API-REST

No siempre disponemos de una API tan elaborada como la presentada en la sección anterior para acceder a los datos de Twitter. Lo que sí es habitual es disponer al menos de una API basada en la arquitectura software de transferencia de estado representacional (REST por sus siglas en inglés, *REpresentational State Transfer*).

El protocolo API-REST define un método sencillo para recibir y enviar datos en cualquier formato, habitualmente XML o JSON, bajo el protocolo HTTP. Para ello se define un pequeño número de funciones para manipular la información, habitualmente POST, GET, PUT y DELETE; en este capítulo nos centraremos en GET, que nos permite recuperar información desde el servidor. La idea básica en REST es la obtención de recursos utilizando una sintaxis sencilla, donde cada recurso y cada elemento de búsqueda tienen un identificador único, que se comunica a través de la URI. Python proporciona varias bibliotecas para realizar peticiones REST.

En este capítulo usaremos la biblioteca `requests`, que ya empleamos en el capítulo anterior para descargar ficheros, y que nos ofrece en particular la función `get`, que recibe como parámetro la dirección del recurso y un diccionario con las opciones de búsqueda (las claves identifican la opción y los valores el valor que queremos que tomen) y devuelve una respuesta (un objeto de clase `Response`). Si los datos que estamos manejando son de tipo JSON podemos solicitarle dicho objeto a la respuesta directamente con la función `json`.

Ejemplo: API de Google Maps

Supongamos que dentro de nuestro programa queremos conocer cuánto se tarda por carretera entre dos localidades. Para obtener esta información utilizaremos:

- La biblioteca `requests` de Python para interactuar con la API-REST de Google

```
>>> import requests
```

- El propio API-REST de Google. Para acceder a él utilizaremos la URL y los parámetros que especifican el origen, el final y el método que queremos usar, en este caso por carretera:

```
>>> URL='http://maps.googleapis.com/maps/api/directions/json'
>>> params = dict(
      origin='Madrid,Spain',
      destination='Barcelona,Spain',
      mode='driving'
    )
```

Ahora estamos listos para hacer la petición, que en este caso es un GET:

```
>>> resp = requests.get(url=URI, params=params)
```

Convertimos la respuesta a formato JSON, para acceder más cómodamente:

```
>>> data = resp.json()
```

Finalmente, tras examinar la variable de tipo diccionario `data`, que contiene gran cantidad de información, encontramos la forma de acceder a la duración de la primera ruta propuesta por Google Maps:

```
>>> print(
      data['routes'][0]['legs'][0]['duration']['text'])
```

que nos mostrará un resultado como: `5 hours 59 mins`.

La mayor parte de las redes sociales y sistemas de compartición de archivos (Facebook, YouTube…) disponen de su propia API-REST. En cada caso habrá que consultar la información del servicio web al que queramos acceder para conocer qué parámetros requiere.

Ejemplo: API de OMDB

Para ilustrar el uso de esta biblioteca vamos a obtener información de películas de OMDB API (`http://www.omdbapi.com/`), una base de datos abierta con información sobre películas y series. En primer lugar, solicitaremos una clave gratuita en la pestaña "API Key" y almacenaremos la clave que recibamos en la variable `clave`. Esta clave la indicaremos en las consultas añadiendo `apikey=` en la URI:

```
>>> clave = "XXX"
>>> uri = "http://www.omdbapi.com/?apikey=" + clave
```

Así, tendremos como base a partir de la cual hacer una consulta a la URI:

```
http://www.omdbapi.com/?apikey=XXX
```

Para realizar búsquedas usaremos la función `get` usando la opción `s` para búsquedas con el valor `The Matrix` y la opción `type` con valor `movie` para evitar series:

```
>>> r = requests.get(uri, {"s" : "The Matrix", "type" : "movie"})
```

Esta búsqueda se correspondería con la siguiente URI, donde Python nos ha ahorrado preocuparnos por el formato:

```
http://www.omdbapi.com/?apikey=XXX&s=The%20Matrix&type=movie
```

Si ahora investigamos el JSON obtenido:

```
>>> r.json()
```

veremos que tenemos un objeto JSON con un elemento `Response` que indica que la solicitud ha devuelto un valor correcto y un elemento `Search` que contiene una lista de objetos JSON con las distintas películas encontradas (solo presentamos el primer resultado para facilitar la lectura):

```
{'Response': 'True',
 'Search': [{'Poster': 'https://ia.media-
imdb.com/images/M/MV5BNzQzOTk3OTAtNDQ0Zi00ZTVkLWI0MTEtMDllZjNkYzNjNTc
4L2ltYWdlXkEyXkFqcGdeQXVyNjU0OTQ0OTY@._V1_SX300.jpg',    'Title':
'The Matrix',    'Type': 'movie',    'Year': '1999',    'imdbID':
'tt0133093'},
...]}
```

Podemos usar ahora el identificador de la primera película que hemos encontrado para obtener más información sobre la película. Para ello usamos la opción `i`, que nos devuelve información sobre una película dado su identificador:

```
>>> r = requests.get(uri, {"i" : "tt0133093"})
>>> r.json()
```

```
{'Actors': 'Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo
Weaving',  'Awards': 'Won 4 Oscars. Another 34 wins & 48
nominations.',  'BoxOffice': 'N/A',  'Country': 'USA',  'DVD': '21
Sep 1999',  'Director': 'Lana Wachowski, Lilly Wachowski',  'Genre':
'Action, Sci-Fi',  'Language': 'English',  'Metascore': '73',
'Plot': 'A computer hacker learns from mysterious rebels about the
true nature of his reality and his role in the war against its
controllers.',  'Poster': 'https://ia.media-
imdb.com/images/M/MV5BNzQzOTk3OTAtNDQ0Zi00ZTVkLWI0MTEtMDllZjNkYzNjNTc
4L2ltYWdlXkEyXkFqcGdeQXVyNjU0OTQ0OTY@._V1_SX300.jpg',  'Production':
'Warner Bros. Pictures',  'Rated': 'R',  'Ratings': [{'Source':
'Internet Movie Database', 'Value': '8.7/10'},   {'Source': 'Rotten
Tomatoes', 'Value': '87%'},   {'Source': 'Metacritic', 'Value':
'73/100'}],  'Released': '31 Mar 1999',  'Response': 'True',
```

```
'Runtime': '136 min',  'Title': 'The Matrix',  'Type': 'movie',
'Website': 'http://www.whatisthematrix.com',  'Writer': 'Lilly
Wachowski, Lana Wachowski',  'Year': '1999',  'imdbID': 'tt0133093',
'imdbRating': '8.7',  'imdbVotes': '1,406,754'}
```

REFERENCIAS

- Matthew A. Rusell y Mikhail Klassen. Mining the social web. O'Reilly Media (tercera edición), 2018.
- Leonard Richardson, Mike Amundsen y Sam Ruby. Restful web APIs. O'Reilly Media, 2013.
- Mark Massé. REST API design rulebook. O'Reilly Media, 2011.
- Página de ayuda de Twitter (accedida en junio de 2018). https://help.twitter.com/es.
- Documentación del objeto `Locations` (accedida en junio de 2018). `https://developer.twitter.com/en/docs/tweets/enrichments/overview/profile-geo`.
- Documentación del objeto `GeoJSON` (accedida en junio de 2018). `http://geojson.org/`.
- Documentación del API Tweepy (accedida en junio de 2018). `http://docs.tweepy.org/en/v3.5.0/api.html`.
- Documentación del objeto `Entities` (accedida en junio de 2018). `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/entities-object`.
- Documentación del objeto `Rule` (accedida en junio de 2018). `http://support.gnip.com/enrichments/matching_rules.html`.

4 MONGODB

INTRODUCCIÓN

En los capítulos anteriores hemos visto numerosas fuentes de datos, como páginas webs, redes sociales o ficheros CSV. Toda esta información, antes de ser analizada, debe ser almacenada adecuadamente, y a ello se dedica este capítulo en el que hablaremos de la base de datos NoSQL más popular: MongoDB.

MongoDB es una base de datos de las denominadas *orientadas a documento*. El nombre indica que la noción de *fila*, usual en las bases de datos SQL, se sustituye aquí por la de documento, en particular por la de documento en formato JSON. Esto nos permitirá almacenar información compleja sin tener que pensar en cómo representar esta información en un formato de tablas, lo que sería obligado en el caso de SQL.

Otra característica de MongoDB es que las *colecciones*, que es como llamaremos a los conjuntos de documentos, no tienen ningún esquema prefijado. Es decir, un documento puede contener unas claves mientras que el documento siguiente, dentro de la misma colección, puede tener una estructura completamente diferente.

Por ejemplo, pensemos en el catálogo de una tienda de ropa, donde cada tipo de prenda tendrá unas características distintas. En SQL esto nos forzaría a crear una tabla para cada tipo de ropa, ya que en las bases de datos relacionales tenemos que definir de antemano, antes de empezar a guardar los datos, la estructura de cada tabla, en particular el nombre y el tipo de cada columna. En cambio, en MongoDB podemos simplemente crear una colección *catálogo*, e ir insertando documentos

JSON con distinta estructura según la prenda concreta. Esta característica nos recuerda a una de las famosas V que definen Big Data, la que hace referencia a la *variedad* en el formato de los datos.

Otra de las V es la de *volumen*, que se refiere a la posibilidad de almacenar grandes cantidades de datos en un entorno escalable. MongoDB también atiende a este requerimiento, permitiendo almacenar los datos en un clúster de ordenadores, donde los datos de una misma colección se encuentran repartidos por todo el clúster. Si la colección sigue creciendo, bastará con añadir nuevos ordenadores al clúster para aumentar la capacidad de almacenamiento.

Podemos pensar que tal cantidad de datos puede llevar a una disminución de rendimiento, pero esto no es así, porque MongoDB también está preparado para satisfacer la tercera V de Big Data, la de *velocidad*. La velocidad, sobre todo de lectura de datos, se logra gracias a que las búsquedas se hacen en paralelo en todos los ordenadores, lo que nos proporciona también escalabilidad en el tiempo de acceso.

A continuación, discutiremos si realmente necesitamos una base de datos, y en caso de que así sea, de qué tipo. En el resto del capítulo nos centraremos en MongoDB, comenzando por la importación de datos, para pasar a revisar lo esencial de su poderoso lenguaje de consultas. Finalmente, trataremos también las operaciones de modificación y borrado de datos.

¿DE VERDAD NECESITO UNA BASE DE DATOS? ¿CUÁL?

Las bases de datos son sistemas diseñados para almacenar información de forma segura, facilitando el acceso y la manipulación eficientes. La primera pregunta que podemos (¡y debemos!) hacernos es si realmente necesitamos utilizar una base de datos. Al fin y al cabo, pese a sus ventajas, esto supone instalar un software adicional y conocer sus particularidades, algo que como veremos lleva su tiempo.

Ya conocemos varias formas alternativas de almacenar datos de forma sencilla y de fácil acceso, como por ejemplo los ficheros CSV, XML o Excel. Por ejemplo, podemos hacer *web scraping* de la página del catastro, y grabar la información obtenida en un fichero '*catastro.csv*' para analizarlos con posterioridad desde Python. Esta es, sin duda, una posibilidad, y de hecho la mayor parte de blogs y libros de ciencias de datos con Python trabajan directamente con esta opción.

¿En qué ocasiones nos interesará emplear una base de datos? ¿Y de qué tipo?

Hasta hace pocos años, la segunda pregunta carecía de sentido, porque solo había un tipo: las bases de datos relacionales, que almacenan los datos en tablas y se encargan de mantener la coherencia de los datos almacenados.

También conocidas como *bases de datos SQ*L, por el nombre del lenguaje de consultas que emplean para recuperar información, estas bases de datos, propuestas por Ted Codd alrededor de 1970, han dominado el mundo del almacenamiento y gestión de los datos durante casi 50 años, y continúan haciéndolo. Pocas propuestas tecnológicas en el mundo del software, si es que la hay, pueden presumir de una longevidad similar.

Sin embargo, la llegada de Big Data ha supuesto un cambio en esta tendencia y ha llevado a plantear alternativas al modelo relacional, las llamadas bases de datos NoSQL, de tal forma que la pregunta "¿qué tipo de base de datos debo elegir?" sí tiene sentido. Repasemos algunos factores que debemos considerar para decidir si optamos por utilizar una base de datos, y la influencia que tienen sobre la elección del tipo concreto.

Consultas complejas

Por ejemplo, supongamos que partimos de datos de usuarios de Twitter, que incluyen el nombre de usuario (*user_screen)*, el número de seguidores (*followers*), la fecha de creación de la cuenta (*created_at*) y el número de tweets totales emitidos *(total_tweets)*.

Una instancia de esta tabla (se llama instancia a los valores concretos que toma la base de datos en un momento dato) podría ser:

User_screen	followers	created_at	total_tweets
Bertoldo	4500	23-02-2012	1200
Herminia	6550	19-08-2007	3333
Calixto	0	14-06-2106	15
Melibea	4500	17-05-2016	800
Aniceto	0	01-01-2015	431
…	…	…	…

Supongamos también que queremos relacionar el número de tweets emitidos con el número de seguidores, aunque considerando solo usuarios a partir de 1000 seguidores. En SQL podemos escribir:

```
select followers,AVG(total_tweets)
from usuarios
group by followers
having followers>1000
```

La consulta indica que, a partir de la tabla usuarios (cláusula `from`), se agrupen todas las filas (es decir todos los usuarios) que tienen el mismo número de seguidores (cláusula `group by`). De estos grupos, nos quedamos solo con el campo `followers` junto con la media del campo total tweets dentro de la tabla. Como vemos se trata de una consulta sencilla, que además en una base de datos SQL se ejecutará con gran velocidad, especialmente si hemos creado *índices* adecuados. Igualmente sucederá si escribimos la consulta análoga en MongoDB.

Sin embargo, si deseamos hacer esta consulta a partir de un fichero .CSV previamente cargado desde Python deberemos programarla nosotros. Esto supone más trabajo, mayores posibilidades de equivocarnos y también menor eficiencia. En caso de que estas consultas se den a menudo, o de que la eficiencia sea muy importante, deberemos valorar la posibilidad de utilizar un sistema gestor de bases de datos.

En este caso, el tipo de bases de datos concreto no es importante, siempre y cuando dispongamos de un lenguaje de consultas que nos permita obtener la información deseada de forma rápida y cómoda. El lenguaje SQL es sin duda muy potente, pero el lenguaje de consultas de, por ejemplo, MongoDB, que veremos más adelante en este mismo capítulo, también lo es.

Esquema de datos complejo o cambiante

Otro caso en el que debemos plantearnos la utilización de una base de datos es cuando queremos almacenar información que se encuentra de forma natural repartida entre varios ficheros, cada uno con sus características. Las grandes bases de datos de gestión que se utilizan comúnmente en las empresas son un buen ejemplo: una tabla de clientes, otra de pedidos, otra tabla con lugares de distribución, una más de facturación que combina datos de varias de las anteriores, etc. En estos casos no hay duda de que un gestor de bases de datos es la solución imprescindible, y en particular de bases de datos relacionales, que nos ofrecen la posibilidad de tener los datos repartidos entre múltiples tablas, e incluyen mecanismos sencillos para conectar las tablas entre sí (como las *claves ajenas*).

Si no pensamos en varias tablas, sino en una que contenga elementos con estructura compleja y cambiante, por ejemplo cuando queramos representar páginas web, las bases de datos más adecuadas son las orientadas a documento, como es el caso de MongoDB. En este caso el formato CSV no es una alternativa, ya que cada línea de texto debe tener las mismas columnas y deberemos emplear una base de datos NoSQL. Como veremos, MongoDB permite que un documento (el equivalente a una fila en CSV/SQL) tenga unas claves (el equivalente a columnas) diferentes al siguiente.

Gran volumen de datos

Finalmente, una última razón para considerar la utilización de un sistema gestor de base de datos es disponer de una cantidad realmente grande de datos, tantos que o bien no sea posible almacenarlos en un solo ordenador, o que, aun siendo posible, su manejo se complique en exceso o presente severos problemas de eficiencia. Nos encontramos de nuevo ante el volumen de Big Data.

En cuanto al tipo de bases de datos, tanto las relacionales como NoSQL son capaces de almacenar y gestionar de forma eficiente la mayor parte de los conjuntos de datos con los que podamos trabajar. Solo cuando nos encontremos ante una cantidad realmente grande, o que podamos prever que va a llegar a serlo, las relacionales dejan de ser una alternativa y no existe otra opción que elegir una base de datos NoSQL.

ARQUITECTURA CLIENTE-SERVIDOR DE MONGODB

Casi todas las bases de datos siguen un modelo de arquitectura conocido como cliente-servidor. El *servidor* es el programa que accede directamente a los datos y ofrece este servicio a los clientes. Por su parte un *cliente* es un programa que accede al servidor para solicitarle datos o pedir que haga modificaciones sobre la base de datos. Muchos clientes pueden estar conectados al mismo servidor.

En este capítulo vamos a considerar dos tipos de clientes: la propia consola que viene por defecto con la base de datos y el cliente Python incluido en la biblioteca `pymongo`.

Acceso al servidor

Un error común en todas las bases de datos, y en particular en Mongo, es intentar utilizar un cliente sin tener el servidor activo. Para ver si el servidor está activo lo más sencillo es acceder el cliente consola, y ver si logra conectar. Para ello abrimos un

terminal (Linux/Mac OS) o un 'Símbolo de Sistema' en Windows e intentamos acceder al cliente tecleando simplemente `mongo`.

Si obtenemos una respuesta como

```
MongoDB consola version v3.4.10
connecting to: mongodb://127.0.0.1:27017
…
exception: connect failed
```

será que el cliente no ha logrado conectar con el servidor. La instrucción `mongodb://127.0.0.1:27017` nos indica que el cliente está "buscando" el servidor como alojado en esta máquina (127.0.0.1 es un número especial que representa una conexión local), y dentro de ella al puerto 27017. Este es el puerto por defecto en el que el servidor MongoDB ofrece el servicio de acceso a los datos. Si en lugar de ese puerto sabemos que el servidor está accesible a través de otro puerto, digamos el 28000, podemos iniciar el cliente con `mongo -port 28000`. Igualmente, si el servidor no está alojado en el servidor local, sino que es un servicio remoto, por ejemplo proporcionado por el servicio *Atlas*, ofrecido por MongoDB para la creación de clústeres alojados en la nube (en particular en *Amazon AWS*), tendremos que incluir tras la llamada a `mongo` la URI proporcionada por el servicio.

En nuestro caso, si cuando tecleamos simplemente `mongo` el sistema se conecta con éxito, es que ya disponemos de un servidor de datos, posiblemente porque el servidor está incluido en el servicio de arranque del sistema operativo.

Puesta en marcha del servidor

En cualquier caso, no está de más que sepamos arrancar nuestro propio servidor. El primer paso es disponer de una carpeta de datos vacía. Es allí donde el servidor alojará los datos que insertemos. Por supuesto, la siguiente vez podremos arrancar el servidor sobre esta misma carpeta, y tendremos a nuestra disposición los datos que allí quedaron almacenados.

Para iniciar el servidor vamos a un terminal del sistema operativo y tecleamos

```
mongod -port 28000 -dbpath C:\datos
```

donde `C:\datos` es la carpeta de datos, ya sea vacía inicialmente o con los datos de la vez anterior. Si todo va bien, tras muchos mensajes aparecerá *'waiting for connections on port 28000'* lo que indica que el servidor está listo.

Importante: tras iniciarse el servidor, este terminal queda bloqueado, dedicado a actuar de servidor. Podemos minimizar la ventana y dejarla aparte, pero no interrumpirla con *Ctrl-C*, ni cerrarla, porque pararíamos el servidor. Hay opciones que permiten arrancar el servidor sin que quede bloqueado el terminal, pero de momento conviene usar esta para poder detectar por pantalla las conexiones, errores, etc., de forma sencilla.

Y hablando de otras posibilidades, conviene mencionar que si se desean utilizar transacciones, incluidas a partir de la versión 4 de Mongo pero no discutidas por razones de brevedad en este libro, deberemos inicializar el servidor con otras opciones que podemos encontrar en la (excelente) documentación que proporciona MongDB a través de su página web.

Una vez iniciado el servidor, podremos acceder al cliente de consola tecleando desde el terminal

```
mongo –port 28000
```

Tras algunos mensajes de inicialización llegaremos al *prompt* de la consola de Mongo. El puerto debe coincidir con el utilizado al iniciar el servidor. Si no se pone ninguno al iniciar `mongod`, este elegirá por defecto el 27017.

Podemos preguntar por ejemplo por las bases de datos que ya existen con la instrucción `show databases`:

```
> show databases
admin   0.000GB
config  0.000GB
local   0.000GB
test    0.001GB
```

Aunque no hemos creado ninguna base de datos, MongoDB ya ha creado varias. Por defecto, la consola de MongoDB nos situará en la base de datos `test`. Si queremos cambiar a otra base de datos podemos teclear, por ejemplo

```
> use pruebas
```

Y ya estaremos en la base de datos `pruebas`. Puede que choque al lector, especialmente si está habituado a bases de datos relacionales, el hecho de que accedamos a una base de datos que no hemos creado previamente. En MongoDB todo va a ser así: cuando accedemos a un objeto que no existe, el sistema lo crea. Esto es muy cómodo y rápido, aunque un poco peligroso si nos equivocamos al

teclear porque no nos saldrá error. MongoDB tiene buen carácter y va a ser realmente difícil hacerlo "enfadar", es decir, lograr obtener un mensaje de error.

Si deseamos salir de la consola teclearemos:

```
> quit()
```

Una nota final acerca de la consola: está escrita en *JavaScript*, y admite todas las instrucciones de este lenguaje. Aunque aquí no vamos a aprovechar esta funcionalidad, no está de más saber que es bastante común desarrollar *scripts* que combinen instrucciones de Mongo con instrucciones JavaScript para hacer tratamientos complejos de la información.

BASES DE DATOS, COLECCIONES Y DOCUMENTOS

Para trabajar en MongoDB debemos conocer su terminología, que por otra parte es común a otras bases de datos orientadas a documentos como CouchDB.

En MongoDB, el equivalente a las tablas de bases de datos relacionales serán las *colecciones*. Las colecciones agrupan *documentos*, que serían el correspondiente a *filas* en el modelo relacional. Los documentos se escriben en formato JSON que, como vimos en el capítulo 1, tiene el aspecto

```
{clave1:valor, …., claven:valorn}
```

Las claves representan las columnas y los valores el contenido de la celda. Los valores pueden ser atómicos, como numéricos, booleanos o *strings*, pero también pueden ser arrays o incluso otros documentos JSON. Un ejemplo:

```
{_id:1, nombre:"Berto", contacto: {mail: "berto@jemail.com",
                                   telfs:[ "45612313", 4511]} }
```

Este documento tiene 3 claves: `_id`, que es numérica, `nombre`, de tipo `str`, y `contacto`, que es a su vez un documento JSON con dos claves, el `mail` de tipo `str` y `telfs`, que es un array. Algunas observaciones sobre los documentos JSON en Mongo:

- La clave `_id` es obligatoria y debe ser distinta para todos los documentos de una colección. Si al insertar el documento no la incluimos, Mongo la añadirá por su cuenta, y al hacer las consultas nos encontraremos con algo como `"_id" : ObjectId("5b2b831d61c4b790aa98e968")`. Si incluimos un `_id` repetido, Mongo dará un error.

- En la consola no hace falta poner las comillas en las claves, puesto que las añade Mongo implícitamente. Sí que harán falta si las claves contienen espacios o algunos caracteres especiales.
- Como vemos en el ejemplo, los arrays pueden tener valores de tipos distintos.
- El formato JSON no incluye en su especificación un formato fecha. MongoDB sí lo hace. Por ejemplo, ISODate("2012-12-19T06:01:17.171Z") representa una fecha y una hora en zona horaria "Zulu" (la Z del final), que corresponde a tiempo coordinado universal (UTC) +0, también llamada "la hora de Greenwich".

CARGA DE DATOS

El primer paso es ser capaz de incorporar datos a nuestras colecciones. Vamos a ver dos formas de hacerlo.

Instrucción `insert`

La forma más sencilla de añadir un documento a una colección es a través de la instrucción `insert`. Por ejemplo, desde la consola, comenzamos por entrar en `mongo` (omitiremos este paso de ahora en adelante):

```
> mongo twitter -port 28000
```

La palabra `twitter` tras la llamada a mongo indica que tras entrar en Mongo debe situarse en esta base de datos, que se creará automáticamente si no existe. También podríamos omitir este argumento y después, ya dentro de la consola, escribir `use twitter`. En cuanto al parámetro `-port 28000` recordemos que debe ser el puerto donde está "escuchando" el servidor `mongod`.

Una vez dentro de la consola, podemos escribir:

```
db.tweets.insertOne(
     {_id: 1,
      usuario: {nick:"bertoldo",seguidores:1320},
      texto: "@herminia: hoy, excursión a la sierra con @aniceto!",
      menciones: ["herminia", "aniceto"],
      RT: false}  )
```

Esto hace que se inserte un nuevo documento en la colección `tweets` de la base de datos actual (`twitter`). El documento representa un tweet, con su identificador,

los datos del usuario (*nick* usado en Twitter y número de seguidores), el texto del tweet, un array con los usuarios mencionados, y un valor *RT* indicado si es un retweet.

Si todo va bien, el sistema nos lo indicará, mostrando además el `_id` del documento. Si en algún momento nos equivocamos y queremos borrar la colección podemos escribir:

```
// Ojo: borra toda la colección
db.tweets.drop()
```

Ya dijimos que MongoDB es "tímido", así que no nos pedirá confirmación y borrará la colección completa sin más, por lo que conviene utilizar esta instrucción con cautela.

Aunque la consola es cómoda y adecuada para hacer pruebas, normalmente querremos integrar el procedimiento de inserción en nuestros programas en Python. Con este fin podemos utilizar la biblioteca `pymongo`.

Comenzamos por cargar la biblioteca y establecer una conexión con el servidor:

```
>>> from pymongo import MongoClient
>>> client = MongoClient('mongodb://localhost:28000/')
```

Comenzamos importando la clase `MongoClient` de `pymongo`, y luego establecemos la conexión, que queda almacenada en la variable `client`. Esta variable hará de puente entre el cliente y el servidor, y en el resto del capítulo asumiremos que existe sin declararla de nuevo.

Ahora podemos acceder a la base de datos `twitter`, y dentro de ella a la colección `tweets`.

```
>>> db = client['twitter']
>>> tweets = db['tweets']
```

Finalmente procedemos a la inserción del documento. Para facilitar la lectura, primero asignamos el documento a una variable intermedia:

```
>>> tweet = {
     '_id':2,
     'usuario': {'nick':"herminia",'seguidores':5320},
     'texto':"RT:@herminia: hoy,excursión a la sierra con @aniceto!",
     'menciones': ["herminia", "aniceto"],
```

```
    'RT': True,
    'origen': 1 }
>>> insertado = tweets.insert_one(tweet)
>>> print(insertado.inserted_id)
```

En este caso el tweet es un retweet (reenvío) del tweet anterior. Un detalle de sintaxis es que en la consola de Mongo los valores booleanos se escriben en minúsculas, mientras que en Python se escribe la primera letra en mayúscula.

En nuestra aplicación hemos decidido que, cuando un tweet sea un retweet, además de indicarlo en la clave RT, apuntaremos el `_id` del tweet original en la clave *origen*. Esto nos muestra que documentos distintos de la misma colección pueden tener claves diferentes.

Importación desde ficheros CSV o JSON

MongoDB incluye dos herramientas para importar y exportar datos: `mongoimport` y `mongoexport`. Ambas se usan desde la línea de comandos (es decir, no dentro de la consola) y son una excelente herramienta para facilitar la comunicación de datos.

Por ejemplo, si hemos descargado tweets sobre la final de la copa del mundo 2018, y queremos incorporarlo a una colección `final` de la base de datos `worldcup18`, podemos escribir:

```
mongoimport --db worldcup18 --collection final --file final.json
```

Donde `final.json` es el fichero que contiene los tweets. En el caso de que el fichero sea de tipo CSV, hay que indicarlo explícitamente con el parámetro.

```
mongoimport -d rus18 -c final --type csv --headerline --file final.csv
```

En este caso se indica el nombre de la base de datos (`rus18`) y de la colección (`final`) con los parámetros abreviados `-d` y `-c`. Además, se avisa a `mongoimport` de que la primera fila del fichero es la cabecera. Esto es importante porque se utilizarán los nombres en esta cabecera como claves para los documentos JSON importados.

Por eso, si el fichero CSV no incluye cabecera debemos usar la opción `--fields` y dar la lista de nombres entre comas, o `--fieldFile` seguido del nombre de un fichero de texto con dichos nombres, uno por línea del fichero.

Ejemplo: inserción de tweets aleatorios

Tras ejecutar el código anterior tenemos ya dos tweets en la colección tweets. Para tener un conjunto mayor y utilizarlo en el resto del capítulo vamos a generar de forma aleatoria 100 tweets al azar desde Python. El código para lograr este programa empieza estableciendo la conexión, seleccionando la base de datos y la colección y asegurándose de que la colección está vacía (`drop`):

```
>>> from pymongo import MongoClient
>>> import random
>>> import string
>>> client = MongoClient('mongodb://localhost:28000/')
>>> db = client['twitter']
>>> tweets = db['tweets']
>>> tweets.drop()
```

Además de la biblioteca `pymongo` para conectar con MongoDB, empleamos `random`, para generar los valores aleatorios, y `str` para las operaciones que permiten generar el texto del tweet.

A continuación, fijamos los nombres y seguidores de cuatro usuarios inventados y el número de tweets a generar (100)

```
>>> usuarios = [("bertoldo",1320),("herminia",5320),
            ("aniceto",123),("melibea",411)]
>>> n = 100
```

Finalmente, el bucle que genera e inserta los tweets:

```
>>> for i in range(1,n+1):
    tweet = {}
    tweet['_id'] = i
    tweet['text'] =
         ''.join(random.choices(string.ascii_uppercase, k=10))
    u = {}
    u['nick'], u['seguidores'] = random.choice(usuarios)
    tweet['usuario'] = u
    tweet['RT'] = i>1 and random.choice([False,True])
    if tweet['RT'] and i>1:
           tweet['origen'] = random.randrange(1, i)
    m =  random.sample(usuarios,
                random.randrange(0, len(usuarios)))
    tweet['mentions'] = [nick for nick,_ in m]
    tweets.insert_one(tweet)
```

Para cada tweet se genera un diccionario vacío (`tweet={}`) que se va completando, primero con el `_id`, después con el texto formado como la sucesión de 10 caracteres aleatorios en mayúscula, luego con los datos del usuario, que se eligen al azar del array `usuarios`.

El marcador `RT` se elige al azar, excepto para el primer tweet, que nunca puede ser retweet. Si `RT` es `true`, se añade la clave `origen` con el `_id` del de uno cualquiera de los tweets anteriores, simulando que ese tweet anterior es el que se está reemitiendo. En el caso del retweet, el texto debería ser de la forma "RT: "+text, con text el texto original. Nuestra simulación no llega a tanto, y se limita a añadir "RT:" al texto aleatorio generado para el tweet.

Finalmente, para las menciones se toma una muestra de los usuarios y se añaden sus nicks a la lista `mentions` (aunque en nuestra pobre simulación las menciones no salen en el tweet).

Para comprobar que todo ha funcionado correctamente podemos ir a la consola de Mongo y ejecutar el siguiente comando, que debe devolver el valor 100:

```
> db.tweet.count()
```

CONSULTAS SIMPLES

Ya conocemos un par de mecanismos sencillos para introducir documentos en nuestra base de datos. Lo siguiente que tenemos que hacer es ser capaces de extraer información, es decir, de hacer consultas. En MongoDB se distinguen entre las consultas simples que vamos a ver en esta sección y las consultas agregadas o de agrupación que veremos en la sección siguiente.

Las siguientes subsecciones se desarrollan dentro de la consola de Mongo. Antes de terminar el apartado veremos cómo se adapta la notación para su uso desde `pymongo`.

`find`, `skip`, `limit` y `sort`

La forma más simple de ver el contenido de una colección desde dentro de la consola de Mongo es simplemente:

```
> db.tweets.find()
```

Esto nos mostrará los 20 primeros documentos de la colección tweets:

```
{ "_id" : 1, "text" : "GKAXRKDQKV", "usuario" : { "nick" : "herminia", "seguidores" : 5320 }, "RT" : false, "mentions" : [ "herminia", "melibea", "bertoldo" ] }

{ "_id" : 2, "text" : "IWGXXFPHSI", "usuario" : { "nick" : "bertoldo", "seguidores" : 1320 }, "RT" : false, "mentions" : [ "aniceto", "herminia", "bertoldo" ] }

…
```

Si la colección tiene más de 20 documentos, podemos teclear `it` para ver los 20 siguientes y así sucesivamente.

El formato en el que se muestran los documentos no es demasiado agradable, y en el caso de JSON puede ser muy difícil de entender. La consola nos permite mejorar esto, a costa de que cada documento ocupe más en vertical.

```
> db.tweets.find().pretty()

{
        "_id" : 1,
        "text" : "GKAXRKDQKV",
        "usuario" : {
                "nick" : "herminia",
                "seguidores" : 5320
        },
        "RT" : false,
        "mentions" : [
                "herminia",
                "melibea",
                "bertoldo"
        ]
}
…
```

La función pretty() “embellece” la salida y la hace más legible. Los documentos se muestran en el mismo orden en el que se insertaron.

Podemos “saltarnos” los primeros documentos con `skip`. Por ejemplo, para ver todos los documentos, pero comenzando a partir del segundo:

```
> db.tweets.find().skip(1).pretty()
```

Esta forma de combinar operaciones componiéndolas mediante el operador punto es muy típica de Mongo y da mucha flexibilidad. Otra función similar es

`limit(n)` que hace que únicamente se muestren los *n* primeros documentos. Por ejemplo, para ver solo los tweets que ocupan las posiciones 6 y 7 podemos emplear:

```
> db.tweets.find().skip(5).limit(2).pretty()
```

El orden en el que se muestran los documentos es el de inserción. Para mostrarlos en otro orden, lo mejor es emplear la función `sort`(). Esta función recibe como parámetro un documento JSON con las claves que se deben usar para la ordenación, seguidas por +1 y si se desea una ordenación ascendente (de menor a mayor), o −1 si se desea que sea descendente (de mayor a menor).

Por ejemplo, para mostrar los tweets comenzando desde el de mayor `_id`, podríamos escribir:

```
> db.tweets.find().sort({_id:-1}).pretty()
{
    "_id" : 100,
    "text" : "BZIVQDRSDU",
    "usuario" : {
                    "nick" : "aniceto",
                    "seguidores" : 123
            },
   "RT" : false,
   "mentions" : [
                    "melibea",
                    "aniceto"
            ]
}
…
```

Supongamos que queremos ordenar por número de seguidores, de mayor a menor. La clave `seguidores` aparece dentro de la clave `usuario`. Para indicar que queremos ordenar por este valor usaremos:

```
> db.tweets.find().sort({"usuario.seguidores":-1})

{ "_id" : 1, "text" : "GKAXRKDQKV", "usuario" : { "nick" :
"herminia", "seguidores" : 5320 }, "RT" : false, "mentions" : [
"herminia", "melibea", "bertoldo" ] }
…
```

Esta forma de componer claves, mediante el punto, es muy cómoda y se utiliza mucho en MongoDB para “navegar” los documentos.

`sort` también permite que se ordene por varias claves. Por ejemplo, si queremos ordenar primero por el número de seguidores de forma descendente, y luego, para los tweets de usuarios con el mismo tweet, por el `_id` también de forma descendente, podemos escribir:

```
> db.tweets.find().sort({"usuario.seguidores":-1, _id:-1})
```

Cuando se combina con otras funciones como `limit` o `skip`, `sort` siempre se ejecuta en primer lugar, sin importar el orden en el que se escriba. Si queremos encontrar el tweet con mayor `_id` podemos escribir:

```
> db.tweets.find().sort({_id:-1}).limit(1)
```

Pero también:

```
> db.tweets.find().limit(1).sort({_id:-1})
```

que dará el mismo resultado, aunque sin duda resulta menos legible.

Un apunte final sobre `sort`: en grandes colecciones, puede ser tremendamente lento. La mejor forma de acelerar este tipo de consultas es disponer de un índice. Un índice es una estructura que mantiene una copia ordenada de una colección según ciertos criterios. En realidad, no se trata de una copia de la colección como tal, lo que sería costosísimo en términos de espacio; tan solo se guarda un "puntero" o señal a cada elemento de la colección real.

Si sabemos, por ejemplo, que vamos a repetir la consulta anterior a menudo, podemos crear un índice para acelerarla con:

```
> db.tweets.createIndex({"usuario.seguidores":-1, _id:-1})
```

La instrucción, que únicamente debe ejecutarse una vez, no tiene ningún efecto aparente, pero puede hacer que una consulta que tardaba horas pase a requerir pocos segundos.

Por supuesto, la magia no existe, al menos en informática, y tan maravillosos resultados tienen un coste: los índices aceleran las consultas, pero retrasan ligeramente las inserciones, modificaciones y borrados. Esto es así porque ahora cada vez que, por ejemplo, se inserta un documento, también hay que apuntar su lugar correspondiente en el índice. Por ello no debemos crear más índices de los necesarios y únicamente usarlos para acelerar consultas que realmente lo precisen.

Estructura general de `find`

La función `find`, que hemos mencionado ya en el apartado anterior, es la base de las consultas simples en Mongo. Su estructura general es:

```
> find({filtro},{proyección})
```

El primer parámetro corresponde al filtro, que indicará qué documentos se deben mostrar. El segundo, la proyección, indicará qué claves se deben mostrar de cada uno de estos documentos. Como hemos visto en el apartado anterior, ambos son opcionales; si se escribe simplemente `find()` se mostrarán todos los documentos y todas sus claves.

Proyección en `find`

Si se incluye solo un argumento en `find`, Mongo entiende que se refiere a la selección. Por ello, si queremos incluir solo la proyección, la selección deberá aparecer, aunque sea como el documento vacío

```
> find({},{proyección})
```

La proyección puede adoptar 3 formas:

1. { }: indica que deben mostrarse todas las claves. En este caso, normalmente nos limitaremos a no incluir este parámetro, lo que tendrá el mismo efecto.
2. {$clave_1$:1, …, $clave_k$:1 }: indica que solo se muestren las claves $clave_1$…$clave_k$.
3. {$clave_1$:0, …, $clave_k$:0}: indica que se muestren todas las claves menos las claves $clave_1$…$clave_k$.

Por ejemplo, para ver todos los datos de cada tweet excepto los datos del usuario, podemos usar la forma 3:

```
> db.tweets.find({},{usuario:0})

{ "_id" : 1, "text" : "GKAXRKDQKV", "RT" : false, "mentions" : [
"herminia", "melibea", "bertoldo" ] }
{ "_id" : 2, "text" : "IWGXXFPHSI", "RT" : false, "mentions" : [
"aniceto", "herminia", "bertoldo" ] }
…
```

Si solo queremos ver solo el _id del tweet y el texto, podemos utilizar la forma 2:

```
> db.tweets.find({},{_id:1, text:1})

{ "_id" : 1, "text" : "GKAXRKDQKV" }
{ "_id" : 2, "text" : "IWGXXFPHSI" }
…
```

Como se ve en el ejemplo, no se pueden mezclar los unos y los ceros. Solo hay una excepción: el `_id`. Esta clave especial siempre se muestra, aunque no se indique explícitamente en la lista

```
> db.tweets.find({},{text:1})

{ "_id" : 1, "text" : "GKAXRKDQKV" }
{ "_id" : 2, "text" : "IWGXXFPHSI" }
…
```

Por ello, en la forma 2, se admite de forma excepcional el uso de `_id:0`.

```
> db.tweets.find({},{text:1, _id:0})

{ "text" : "GKAXRKDQKV" }
{ "text" : "IWGXXFPHSI" }
…
```

Selección en `find`

Los que conozcan el lenguaje de consultas SQL pueden pensar en el primer argumento de `find`, la selección, como en el equivalente de la cláusula `where`. Veamos ahora sus principales posibilidades.

IGUALDAD

La primera y más básica forma de seleccionar documentos es buscar por valores concretos, es decir, filtrar con criterios de igualdad. Por ejemplo, podemos querer ver tan solo los textos de tweets que son retweets:

```
> db.tweets.find({RT:true}, {text:1,_id:0})

{ "text" : "RT: UFBFDYKXUK" }
{ "text" : "RT: XTCDXTNIVN" }
…
```

El primer argumento selecciona solo los tweets con el indicador `RT` a `true`, mientras que el segundo indica que de los documentos que verifican esto solo se debe mostrar el campo `text`. Podemos refinar el filtrado indicando que solo queremos retweets efectuados por Bertoldo:

```
> db.tweets.find({RT:true, 'usuario.nick':'bertoldo'},{text:1,_id:0})
```

La coma que separa RT y ‘usuario.nick’ se entiende como una conjunción: busca tweets que sean retweets y cuyo nick de usuario corresponda a ‘bertoldo’. Si lo que deseamos es contar el número de documentos que son retweets realizados por Bertoldo, podemos usar la función count:

```
> db.tweets.find({RT:true, 'usuario.nick':'bertoldo'}).count()
15
```

En este caso no hemos incluido el segundo argumento de find, la proyección, porque no varía el número de documentos seleccionados, y por tanto no influye en el resultado.

OTROS OPERADORES DE COMPARACIÓN Y LÓGICOS

La siguiente tabla muestra otros operadores que pueden utilizarse para comparar valores, aparte de la igualdad ya vista:

Operador	Selecciona documentos tales que…
$gt	La clave debe tener un valor estrictamente mayor al indicado
$gte	La clave debe tener un valor mayor o igual al indicado
$lt	La clave debe tener un valor estrictamente menor al indicado
$lte	La clave debe tener un valor menor o igual al indicado
$eq	La clave debe tener el valor indicado
$ne	La clave debe contener un valor distinto del indicado
$and	Verifican todas las condiciones indicadas
$or	Verifican alguna de las condiciones indicadas
$not	No cumplen la condición indicada
$nor	No cumplen ninguna de las condiciones indicadas

Por ejemplo, si queremos contar el total de tweets no emitidos por Bertoldo, podemos utilizar el operador $ne:

```
> db.tweets.find({'usuario.nick':{$ne:'bertoldo'}}).count()
71
```

Otra expresión equivalente se obtiene al restar al total el número de tweets emitidos por Bertoldo.

```
> db.tweets.find().count() -
  db.tweets.find({'usuario.nick':'bertoldo'}).count()
71
```

También podemos usar estos operadores para indicar un rango de valores. Por ejemplo, queremos obtener los tweets de usuarios que tienen entre 1000 y 2000 seguidores (ambos números excluidos):

```
> db.tweets.find({'usuario.seguidores':{$gt:1000, $lt:2000}})
{ "_id" : 99, "text" : "BLIBHOBCGN", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : false, "mentions" : [
"aniceto" ] }
{ "_id" : 97, "text" : "RT: RMXRNHWGJZ", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : true, "origen" : 8,
"mentions" : [ ] }
…
```

Como se ve en el ejemplo, cuando hay varias condiciones sobre la misma clave se agrupan en un mismo lado derecho {'usuario.seguidores':{$gt:1000, $lt:2000}}. Esto es necesario porque un documento JSON de Mongo no puede contener la misma clave repetida dos o más veces, es decir, escribir {'usuario.seguidores':{$gt:1000}, 'usuario.seguidores':{$lt:2000}} es incorrecto y daría lugar bien a errores, o bien a comportamientos inesperados (por ejemplo en la consola solo se tendría en cuenta la segunda condición).

Las condiciones se pueden agrupar usando los operadores $not, $and, $or y $nor. Por ejemplo, si queremos tweets escritos ya sea por Bertoldo o por Herminia podemos escribir:

```
> db.tweets.find({'$or':[{'usuario.nick':"bertoldo"},
                       {'usuario.nick':"herminia"}]})

{ "_id" : 1, "text" : "GKAXRKDQKV", "usuario" : { "nick" :
"herminia", "seguidores" : 5320 }, "RT" : false, "mentions" : [
"herminia", "melibea", "bertoldo" ] }
```

```
{ "_id" : 2, "text" : "IWGXXFPHSI", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : false, "mentions" : [
"aniceto", "herminia", "bertoldo" ] }
…
```

Podría pensarse que esta consulta es incoherente con lo que hemos dicho anteriormente, porque la clave `'usuario.nick'` aparece dos veces. Sin embargo, no lo es, porque la clave aparece en dos documentos distintos. Los operadores `$and`, `$or` y `$nor` llevan en su lado derecho un array de documentos que pueden considerarse independientes entre sí.

ARRAYS

Las consultas sobre arrays en Mongo son muy potentes y flexibles, pero también generan a menudo confusión. El principio inicial es fácil: si un documento incluye por ejemplo `a:[1,2,3,4]` es lo mismo que si incluyera a la vez `a:1`, `a:2`, `a:3` y `a:4`. Por ello si queremos ver la clave `mentions` de aquellos tweets que mencionan a Aniceto, nos bastará con escribir:

```
> db.tweets.find({mentions:"aniceto"}, {mentions:1})
{ "_id" : 2, "mentions" : [ "aniceto", "herminia", "bertoldo" ] }
{ "_id" : 3, "mentions" : [ "aniceto" ] }
{ "_id" : 5, "mentions" : [ "bertoldo", "herminia", "aniceto" ] }
```

La clave `mentions` es un array y la selección indica "selecciona aquellos documentos en los que `mentions`, o bien sea 'aniceto', o bien sea un array que contiene 'aniceto' ".

Esto nos permite seleccionar documentos cuyos *arrays* contienen elementos concretos de forma sencilla. Igualmente podemos preguntar por los tweets que no mencionan a 'aniceto':

```
> db.tweets.find({'mentions':{$ne:"aniceto"}}, {mentions:1})
{ "_id" : 1, "mentions" : [ "herminia", "melibea", "bertoldo" ] }
{ "_id" : 4, "mentions" : [ "bertoldo", "melibea" ] }
```

Sin embargo, también tiene algunos resultados un tanto desconcertantes. Consideremos el siguiente ejemplo:

```
> db.arrays.drop()
> db.arrays.insert({a:[10,20,30,40]})
> db.arrays.find({a:{$gt:20,$lt:30}})

{ "_id" :ObjectId("5b2f8c8080115a9b4011dd8c"), a:[ 10, 20, 30, 40 ] }
```

La consulta parece preguntar si hay un elemento mayor que 20 y menor que 30. Parece no haber ninguno, pero sin embargo la consulta ha tenido éxito. ¿Qué ha ocurrido? Pues que, en efecto el array `a` tiene un valor mayor que 20 (por ejemplo 30) y otro menor que 30 (por ejemplo 20). Es decir, cada condición de la selección se cumple para un elemento diferente del array `a`.

¿Podemos lograr que se apliquen las dos condiciones al mismo elemento? Para esto existe un operador especial, `$elemMatch`:

```
> db.arrays.find({a:{$elemMatch:{$gt:20,$lt:30}}})
```

En este caso no obtenemos respuesta, porque ningún elemento del array está entre 20 y 30.

Existen otros operadores para arrays, como `$all` que selecciona documentos con una clave de tipo array que contenga (al menos) todos los elementos especificados en una lista, `$in` que busca que el array del documento tenga al menos un elemento de una lista, o `$nin`, que requiere que cierta clave de tipo array no tenga ninguno de los elementos indicados. También se pueden hacer consultas con condiciones sobre la longitud del array, por ejemplo la siguiente que selecciona los tweets con al menos 3 menciones:

```
> db.tweets.find({'mentions':{$size:3}}).count()
22
```

$EXISTS

Como hemos visto, diferentes documentos de la misma colección pueden tener claves diferentes. Este operador nos permite seleccionar aquellos documentos que sí tienen una clave concreta. Veamos un ejemplo. Supongamos que queremos mostrar los tweets ordenados por la clave `origen`, de menor a mayor:

```
db.tweets.find().sort({origen:1})

{ "_id" : 1, "text" : "GKAXRKDQKV", "usuario" : { "nick" :
"herminia", "seguidores" : 5320 }, "RT" : false, "mentions" : [
"herminia", "melibea", "bertoldo" ] }
{ "_id" : 2, "text" : "IWGXXFPHSI", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : false, "mentions" : [
"aniceto", "herminia", "bertoldo" ] }
…
```

Observamos que los primeros documentos que se muestran ¡no contienen la clave `origen`! La causa es que cuando una clave no existe, Mongo la considera como

de valor mínimo, y por tanto estos tweets aparecerán los primeros. Para evitarlo debemos utilizar `$exists`:

```
> db.tweets.find({origen:{$exists:1}}).sort({origen:1})

{ "_id" : 3, "text" : "RT: UFBFDYKXUK", "usuario" : { "nick" :
"melibea", "seguidores" : 411 }, "RT" : true, "origen" : 1,
"mentions" : [ "aniceto" ] }
{ "_id" : 29, "text" : "RT: VGMQCGYLKS", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : true, "origen" : 1,
"mentions" : [ ] }
```

El valor 1 tras `$exists` selecciona solo los documentos que tienen este campo. Un valor 0 seleccionaría solo a los que no lo tienen.

find en Python

Los ejemplos anteriores los hemos presentado desde la consola, por la sencillez e inmediatez que ofrece este cliente. El paso a `pymongo`, y por tanto la posibilidad de emplear todas estas facilidades desde Python, es inmediato. Si lo que se desea es acceder tan solo al primero documento que cumpla los criterios se suele utilizar `find_one`:

```
>>> from pymongo import MongoClient
>>> client = MongoClient('mongodb://localhost:28000/')
>>> db = client['twitter']
>>> tweets = db['tweets']
>>> tweet = tweets.find_one({"usuario.nick":'bertoldo'},
  {'text':1,'_id':0})
>>> print(tweet)
{'text': 'IWGXXFPHSI'}
```

En la consola también se puede usar esta misma función, bajo el nombre de `findOne`, para obtener el primer resultado que cumpla la selección.

En muchos ejemplos veremos que se renuncia al uso de la proyección ya que se puede realizar fácilmente desde Python. Por ejemplo, podemos reemplazar las dos últimas instrucciones por

```
>>> tweet = tweets.find_one({'usuario.nick': "bertoldo"})
>>> print('text: ',tweet['text'])
text:  IWGXXFPHSI
```

Si en lugar de un tweet queremos tratar todos los que cumplan las condiciones indicadas, usaremos directamente `find`, que nos devolverá un objeto de tipo `pymongo.collection.Collection` que podemos iterar con una instrucción `for`:

```
>>> for t in tweets.find({'usuario.nick':"bertoldo",
                          'mentions': "herminia"}):
      print(t['text'])
```

De esta forma, podemos combinar toda la potencia de un lenguaje como Python con las posibilidades de las consultas ofrecidas por Mongo.

Un consejo: siempre que podamos, debemos dejar a la base de datos la tarea de resolver las consultas, evitando la "tentación" de hacer nosotros mismos el trabajo en Python. Por ejemplo, en lugar del código anterior, podríamos pensar en escribir:

```
>>> for t in tweets.find():
    if t['usuario']['nick']=="bertoldo" and
          "herminia" in t['mentions']:
        print(t['text'])
```

Esta consulta devuelve el mismo resultado que la anterior, pero presenta varias desventajas en cuanto a eficiencia:

1. Hace que la colección completa 'viaje' hasta el ordenador donde está el cliente, para hacer a continuación el filtrado.
2. No hará uso de índices, ni de las optimizaciones realizadas de forma automática por el planificador de Mongo.

Por tanto, siempre que sea posible, dejemos a Mongo lo que es de Mongo.

AGREGACIONES

Ya hemos visto cómo escribir una gran variedad de consultas con `find`. Sin embargo, no hemos visto aún cómo realizar operaciones de agregación, es decir, cómo combinar varios documentos agrupándolos según un criterio determinado.

En Mongo, esta tarea es realizada por la función `aggregate`. Realmente `aggregate` es más que una función de agregación: permite realizar consultas complejas que no son posibles con `find`, incluso si no implican agregación.

El *pipeline*

La función `aggregate` se define mediante una serie de etapas consecutivas. Cada etapa tiene que realizar un tipo de operación determinado (hay más de 25 tipos). La forma general es:

```
db.tweet.aggregate([ etapa1, …, etapan])
```

La primera etapa toma como entrada la colección a la que se aplica la función `aggregate`. La segunda etapa toma como entrada el resultado de la primera etapa, y así sucesivamente. A esta estructura es a la que se conoce como '*pipeline de agregación en Mongo*'. A continuación, presentamos las etapas principales.

$group

La etapa "reina" permite agrupar elementos y realizar operaciones sobre cada uno de los grupos. El valor por el que agrupar será el `_id` del documento generado.

Como ejemplo, vamos a contar el número de tweets que ha emitido cada usuario:

```
> db.tweets.aggregate(
        [
         {$group:
                { _id:"$usuario.nick",
                  num_tweets:{$sum:1}
                }
          }
         ]
     )

{ "_id" : "melibea", "num_tweets" : 18 }
{ "_id" : "bertoldo", "num_tweets" : 29 }
{ "_id" : "aniceto", "num_tweets" : 28 }
{ "_id" : "herminia", "num_tweets" : 25
```

Esta consulta solo tiene una etapa, de tipo `$group`. El valor de agrupación es '`usuario.nick`'. Llama la atención que el nombre de la clave venga precedido del valor `$`: esto es necesario siempre que se quiera referenciar una clave en el lado derecho de otra.

Por tanto, la etapa considera todos los elementos de la colección `tweets`, y los agrupa por el *nick* del usuario. Esto da lugar a 4 grupos. Ahora, para cada grupo, se crea el campo `num_tweets`, sumando 1 por cada elemento del grupo. El resultado es el valor buscado.

El uso de `$sum:1` es tan común, que a partir de la versión 3.4 Mongo incluye una etapa que hace esto sin que se necesite escribirlo explícitamente. Se llama `$sortByCount`:

```
> db.tweets.aggregate([
    {$sortByCount: "$usuario.nick"}
 ])

{ "_id" : "bertoldo", "count" : 29 }
{ "_id" : "aniceto", "count" : 28 }
{ "_id" : "herminia", "count" : 25 }
{ "_id" : "melibea", "count" : 18 }
```

En `$group` el atributo `_id` puede ser compuesto, lo que permite agrupar por más de un criterio. Por ejemplo, queremos saber para cada usuario cuántos de sus tweets son originales (`RT:false`) y cuántos retweets (`RT:true`). Podemos obtener esta información así:

```
> db.tweets.aggregate(
      [
       {$group:
              { _id:{nick:"$usuario.nick", RT:"$RT"},
                num_tweets:{$sum:1}
              }
        }
       ]
     )
{ "_id" : { "nick" : "aniceto", "RT" : false }, "num_tweets" : 19 }
{ "_id" : { "nick" : "aniceto", "RT" : true }, "num_tweets" : 9 }
{ "_id" : { "nick" : "melibea", "RT" : false }, "num_tweets" : 10 }
{ "_id" : { "nick" : "melibea", "RT" : true }, "num_tweets" : 8 }
{ "_id" : { "nick" : "bertoldo", "RT" : true }, "num_tweets" : 15 }
{ "_id" : { "nick" : "bertoldo", "RT" : false }, "num_tweets" : 14 }
{ "_id" : { "nick" : "herminia", "RT" : true }, "num_tweets" : 10 }
{ "_id" : { "nick" : "herminia", "RT" : false }, "num_tweets" : 15 }
```

Además de `$sum`, se pueden utilizar otros operadores como `$avg` (media), `$first` (un valor del primer documento del grupo), `$last` (un valor del último elemento del grupo), `$max` (máximo), `$min` (mínimo) y dos operadores especiales: `$push` y `$addToSet`.

`$push` genera, para cada elemento del grupo, un elemento de un array. Para ver un ejemplo, supongamos que para cada usuario queremos recoger todos los textos de sus tweets en un solo array.

```
> db.tweets.aggregate(
      [
       {$group:
               { _id:"$usuario.nick",
                 textos:{$push:"$text"}
               }
        }
       ]
     )

{ "_id" : "melibea", "textos" : [ "RT: UFBFDYKXUK", "BVDZDRGDLP",
"BTSVWZSTVX", … ] }
…
```

`$addToSet` es similar, pero con la salvedad de que no repite elementos, es decir, considera el array como un conjunto.

Para terminar con esta etapa hay que mencionar el "truco" utilizado de forma habitual para el caso en el que se quiera considerar toda la colección como un único grupo. Por ejemplo, supongamos que queremos conocer el número medio de menciones entre todos los tweets de la colección:

```
> db.tweets.aggregate(
      [
       {$group:
               { _id:null,
                 menciones:{$avg:{$size:"$mentions"}}
               }
        }
       ]
     )

{ "_id" : null, "menciones" : 1.46 }
```

La idea es que el valor null (en realidad se puede poner cualquier constante, `0`, `true`, o "`tururú`") se evalúa al mismo valor para todos los documentos, esto es, a `null`. De esta forma todos los documentos pasan a formar un único grupo y ahora se puede aplicar la media del número de menciones.

$match

Esta etapa sirve para filtrar documentos de la etapa anterior (o de la colección, si es la primera etapa). Supongamos que queremos ver el total de tweets por usuario, pero solo estamos interesados en aquellos con más de 20 tweets. Podemos escribir:

```
> db.tweets.aggregate([
    {$sortByCount: "$usuario.nick"},
    {$match: {count:{$gt:20}} }
 ])

{ "_id" : "bertoldo", "count" : 29 }
{ "_id" : "aniceto", "count" : 28 }
{ "_id" : "herminia", "count" : 25 }
```

Tal y como hemos visto en el apartado anterior, la primera etapa genera una salida con 4 documentos, uno por usuario, y cada usuario con dos claves, `_id` que contiene el *nick* del usuario, y `count`, que tiene el total de documentos asociados a ese `_id`. Por eso, la segunda etapa selecciona aquellos documentos que tienen un valor `count` mayor de 20.

$project

Esta etapa se encarga de 'formatear' la salida. A diferencia de la proyección de `find`, no solo permite incluir claves que ya existen sino crear claves nuevas, lo que hace que sea más potente. Por ejemplo, la siguiente instrucción conserva únicamente el campo `usuario` y además crea un nuevo campo `numMentions` que contendrá el tamaño del campo `mentions`, que es un array:

```
> db.tweets.aggregate( [
   {
      $project: {
         usuario: 1,
         _id:0,
         numMentions: {$size:"$mentions"} }}
         ])

{ "usuario" : { "nick" : "herminia", "seguidores" : 5320 },
"numMentions" : 3 }
{ "usuario" : { "nick" : "bertoldo", "seguidores" : 1320 },
"numMentions" : 3 }
…
```

Otras etapas: $unwind, $sample, $out, …

Veamos ahora otras etapas usadas a menudo. En primer lugar `$unwind` "desenrolla" un array, convirtiendo cada uno de sus valores en un documento individual. El resultado se ve mejor si creamos un ejemplo pequeño:

```
> db.unwind.drop()
> db.unwind.insert({_id:1, a:1, b:[1,2,3]})
> db.unwind.insert({_id:2, a:2, b:[4,5]})
> db.unwind.aggregate([{$unwind:"$b"}])

{ "_id" : 1, "a" : 1, "b" : 1 }
{ "_id" : 1, "a" : 1, "b" : 2 }
{ "_id" : 1, "a" : 1, "b" : 3 }
{ "_id" : 2, "a" : 2, "b" : 4 }
{ "_id" : 2, "a" : 2, "b" : 5 }
```

La utilidad de este operador solo se aprecia cuando se combina con otros, como veremos en la siguiente sección.

Otro operador sencillo, pero a veces muy conveniente, es `$sample`, que simplemente toma una muestra aleatoria de una colección. Su sintaxis es muy sencilla:

```
> db.tweets.aggregate(   [ { $sample: { size: 2 } } ] )

{ "_id" : 21, "text" : "DTCWGGMCLH", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : false, "mentions" : [
"melibea", "bertoldo", "herminia" ] }
{ "_id" : 20, "text" : "RT: LWXLFLEXZT", "usuario" : { "nick" :
"bertoldo", "seguidores" : 1320 }, "RT" : true, "origen" : 17,
"mentions" : [ "herminia" ] }
```

El parámetro `size` indica el tamaño de la muestra. Finalmente hay que mencionar otra etapa muy sencilla pero casi imprescindible: `$out`, que almacena el resultado de las etapas anteriores como una nueva colección. Siempre debe ser la última etapa. Por ejemplo:

```
> db.tweets.aggregate( [  { $sample: { size: 3 } },
                      { $out: "minitweets" } ] )
```

crea una nueva colección `minitweets` con una muestra de 3 documentos tomados de forma aleatoria de la colección `tweets`.

Además de los ya vistos existen otras etapas con significado análogo al de las funciones equivalentes en `find`: `$sort`, `$limit` y `$skip`.

$lookup

En MongoDB la mayoría de las consultas afectan a una sola colección. Si nuestra base de datos estuviera en el modelo relacional hubiéramos utilizado dos tablas: una de usuarios y otra de tweets, que se combinarían con operaciones `join` cuando hiciera falta.

En Mongo se prefiere combinar las dos tablas en una sola colección y evitar estas operaciones *join*, que suelen resultar costosas en cuanto a tiempo en las bases de datos NoSQL. Sin embargo, en ocasiones no hay más remedio que combinar dos colecciones en la misma consulta. En estos casos es cuando la etapa `$lookup` tiene sentido.

Supongamos que para cada tweet que es un retweet queremos saber: el _id del retweet, el usuario que lo ha emitido y el usuario que emitió el tweet original. Para entender lo que debemos hacer consideremos un retweet cualquiera:

```
> db.tweets.findOne({RT:true})
{
        "_id" : 3,
        "text" : "RT: UFBFDYKXUK",
        "usuario" : {
                "nick" : "melibea",
                "seguidores" : 411
        },
        "RT" : true,
        "origen" : 1,
        "mentions" : [
                "aniceto"
        ]
}
```

Ya tenemos el `_id` del retweet (3), y el usuario (`melibea`), pero nos falta el nombre del usuario que emitió el tweet original. Para encontrarlo debemos encontrar en la colección `tweets` un documento cuyo `_id` sea el mismo que el que indica la clave `origen`.

La estructura general de `$lookup`:

```
{
   $lookup:
     {
       from: <colección a combinar>,
```

```
       localField: <clave de los documentos origen>,
       foreignField: <clave de los documentos de la colección
                      "from">,
       as: <nombre del campo array generado>
     }
}
```

En nuestro ejemplo la colección `from` será la propia tweets. El `localField` será `origen`, y el `foreignField` la clave `_id` (el del tweet original). En la clave `as` debemos dar el nombre de una nueva clave. A esta clave se asociará un array con todos los tweets cuyo _id coincida con el del retweet.

El ejemplo completo:

```
> db.tweets.aggregate([
{  $match: {RT:true } },
{
   $lookup:
     {
       from: "tweets",
       localField: "origen",
       foreignField: "_id",
       as: "tweet_original"
     }
},
{  $unwind:"$tweet_original"},
{  $project:{_id:"$_id",emitido:"$usuario.nick",
             fuente:"$tweet_original.usuario.nick"}}
])

{ "_id" : 3, "emitido" : "melibea", "fuente" : "herminia" }
{ "_id" : 4, "emitido" : "bertoldo", "fuente" : "melibea" }
{ "_id" : 9, "emitido" : "herminia", "fuente" : "melibea" }
{ "_id" : 11, "emitido" : "bertoldo", "fuente" : "herminia" }
…
```

Hemos necesitado cuatro etapas: la primera para filtrar por RT a `true` la segunda para añadir la información del tweet original, luego desplegamos el array "tweet_original", que sabemos que solo contiene un documento. Finalmente usamos `$project` para formatear la salida.

Ejemplo: usuario más mencionado

Queremos saber cuál es el usuario que ha recibido más menciones dentro de la colección `tweets`, pero teniendo en cuenta solo tweets originales. La idea sería

agrupar por la clave `mentions`, pero es un array, así que tendremos primero que desplegar el array usando `unwind`.

```
db.tweets.aggregate([
    {$match:{"RT":true}},
    {$unwind:"$mentions"},
    {$sortByCount: "$mentions"},
    ])

{ "_id" : "bertoldo", "count" : 15 }
{ "_id" : "herminia", "count" : 14 }
{ "_id" : "aniceto", "count" : 13 }
{ "_id" : "melibea", "count" : 12 }
```

Primero filtramos los tweets para quedarnos solo con los que tienen la clave `RT` a `true` (etapa `$match`). Luego, convertimos cada mención en un solo documento (`$unwind`), y finalmente contamos y ordenamos por el resultado (`$sortByCount`).

VISTAS

Las vistas nos permiten 'nombrar' consultas de forma que la consulta queda almacenada y se ejecuta cada vez que es invocada. Veamos un ejemplo:

```
> db.createView("mencionesOriginales","tweets",
   [
     {$match:{"RT":true}},
     {$unwind:"$mentions"},
     {$sortByCount: "$mentions"},
   ])
```

El primer parámetro es el nombre de la vista a crear, el segundo el nombre de la colección de partida, y finalmente el tercero es un *pipeline* de agregación. El resultado es aparentemente similar a la creación de una nueva colección:

```
> show collections
…
mencionesOriginales
…
```

sobre la que se puede hacer `find`

```
> db.mencionesOriginales.find()
{ "_id" : "bertoldo", "count" : 15 }
{ "_id" : "herminia", "count" : 14 }
{ "_id" : "aniceto", "count" : 13 }
{ "_id" : "melibea", "count" : 12 }
```

Sin embargo, debemos recordar que cada vez que se hace `find` sobre una vista se ejecuta la consulta asociada. Esto hace que la vista cambie al modificarse la colección de partida (`tweets`), y también, por supuesto, que su eficiencia sea menor que la consulta sobre una colección normal ya que implica ejecutar el *pipeline* de agregación asociado.

UPDATE Y REMOVE

Para finalizar, veamos estas dos operaciones que permiten modificar o eliminar documentos ya existentes, respectivamente.

La forma más sencilla de modificar un documento es simplemente reemplazarlo por otro. En este caso `update` tiene dos argumentos: el primero selecciona el elemento a modificar y el segundo es el documento por el que se sustituirá. Veamos un ejemplo basado en la siguiente pequeña colección:

```
> use astronomia
> db.estelar.insert({_id:1, nombre:"Sirio", tipo:"estrella",
                        espectro:"A1V"})
> db.estelar.insert({_id:2, nombre:"Saturno", tipo:"planeta"})
> db.estelar.insert({_id:3, nombre:"Plutón", tipo:"planeta"})
```

Queremos cambiar el tipo de Plutón a “Planeta Enano”. Podemos hacer:

```
> db.estelar.update({_id:3}, {tipo:"planeta enano"})
> db.estelar.find({_id:3}

{ "_id" : 3, "tipo" : "planeta enano" }
```

Tras la modificación se ha perdido la clave ‘nombre’ ¿Qué ha ocurrido? Pues sencillamente que, tal y como hemos dicho, en un *update total*, debemos proporcionar el documento completo, pero solo hemos proporcionado el tipo (y MongoDB ha mantenido el `_id` que es la única clave que no puede modificarse). Para no perder datos deberíamos haber escrito:

```
> db.estelar.update({_id:3}, { nombre:"Plutón",
                               tipo:"planeta enano"})
```

Parece entonces que este tipo de *updates* no son útiles, si nos obliga a reescribir el documento completo. Sin embargo, sí son interesantes cuando, en lugar de usar la

consola utilizamos Python. Veamos el mismo ejemplo, pero a través de pymongo. Empezamos preparando la base de datos:

```
>>> from pymongo import MongoClient
>>> client = MongoClient('mongodb://localhost:28000/')
>>> db = client['astronomia']
>>> estelar = db['estelar']

>>> estelar.drop()
>>> estelar.insert_many([
   {'_id':1,'nombre':"Sirio",'tipo':"estrella", 'espectro':"A1V"},
   {'_id':2,'nombre':"Saturno", 'tipo':"planeta"},
   {'_id':3,'nombre':"Plutón",'tipo':"planeta"}   ] )
```

En la preparación de la base de datos hemos utilizado la función insert_many, que permite insertar un array de documentos en la misma instrucción.

Ahora ya podemos hacer el update total. En el caso de pymongo este tipo de operación lleva el muy adecuado nombre de replace, en este caso particular, replace_one:

```
>>> pluton  = estelar.find_one({'_id':3})
>>> pluton['tipo']  = "planeta enano"
>>> estelar.replace_one({'_id':pluton['_id']},pluton)
```

En este caso, como podemos ver, primero "cargamos" el documento a modificar mediante find_one, lo modificamos, y lo devolvemos a la base de datos a través de replace_one. La diferencia con la consola está en que en ningún momento hemos tenido que escribir el documento entero.

En todo caso, las modificaciones de este tipo se realizan mejor mediante los updates parciales, que mostramos a continuación.

Update parcial

A diferencia del update total, en el parcial en lugar de reemplazar el documento completo, solo se especifican los cambios a realizar.

```
> db.estelar.updateOne( {nombre:"Plutón"},
                    {$set : { tipo: "planeta enano"}})

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

El operador `$set` indica que se va a listar una serie de claves, que deben modificarse con los valores que se indican. La respuesta de Mongo nos indica que ha encontrado un valor con el filtro requerido (`{nombre:"Plutón"}`) y que se ha modificado. Podría ser que no se modificara, por ejemplo, si Mongo comprueba que ya tiene el valor indicado. El valor *matchedCount* nunca valdrá más de 1 en el caso de `updateOne`, llamado `update_one` en `pymongo`, porque esta función se detiene al encontrar la primera coincidencia.

Si se quieren modificar todos los documentos que cumplan una determinada condición, se debe utilizar `updateMany` desde la consola (`update_many` en `pymongo`).

```
> db.estelar.updateMany( {}, {$currentDate : { fecha: true}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

En este caso se seleccionan todos los documentos (filtro `{}`) y se les añade la fecha actual. Para esto, en lugar del operador `$set` utilizamos `$currentDate`, que añade la fecha con el nombre de clave indicado.

Además de `$set` y `$currentDate` hay muchos otros operadores de interés. Por ejemplo `$rename` es muy útil para renombrar claves. Si deseamos que la clave "tipo" pase a llamarse "clase" utilizaremos:

```
> db.estelar.updateMany( {}, { $rename: { "tipo": "clase" } } )
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

También es de interés el operador de modificación `$unset`, que permite eliminar claves existentes. En caso de desear eliminar la clave "espectro" del documento asociado a la estrella "Sirio", podemos escribir:

```
> db.estelar.updateOne( {nombre:"Sirio"},
                      { $unset: { "espectro": true } } )
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Puede chocar la utilización del valor `true`. En realidad, se puede poner cualquier valor, pero no se puede dejar en blanco porque debemos respetar la sintaxis de JSON.

Otro grupo de operadores interesante son los aritméticos: `$inc`, `$max`, `$min`, `$mul`, que permiten actualizar campos. Para ver su funcionamiento supongamos que tenemos una colección de productos con elementos de la forma:

```
> db.productos.insert({_id:"123", cantidad:10, vendido:0})
```

y que queremos registrar una venta, incrementando el valor de la clave "vendido", y decrementando en 1 la cantidad de valores de este producto que tenemos en el almacén.

```
> db.productos.update(
      { _id: "123" },
      { $inc: { almacen: -1, vendido: 1 } }
   )
```

El operador $inc también es muy interesante para actualizar contadores de visitas en páginas web.

Upsert

Supongamos que es importante que aseguremos que en la colección clientes aparece que Bertoldo se ha dado de baja. Podemos escribir:

```
db.clientes.updateOne({nombre:'Bertoldo'},{$set:{baja:true}})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
```

Puede suceder que, aunque Bertoldo haya decidido darse de baja, no conste en nuestras bases de datos, y que por tanto el update anterior no tenga efectos. Sin embargo, incluso en este caso queremos apuntar la baja, por ejemplo, para evitar futuras acciones comerciales con alguien que ha dicho explícitamente que no las desea.

Esta situación, donde queremos modificar el documento si existe y crearlo si no existe, se conoce en MongoDB, y en general en el mundo de las bases de datos, como *upsert*, y se indica añadiendo un parámetro adicional a update:

```
> db.clientes.updateOne({nombre:'Bertoldo'},
                        {$set:{baja:true}},{upsert:true})
   {
           "acknowledged" : true,
           "matchedCount" : 0,
           "modifiedCount" : 0,
           "upsertedId" : ObjectId("5b312810ceb80d4e45f08445")
   }
```

Mongo nos informa de que ningún documento cumple nuestra selección y que ninguno ha sido modificado, pero también nos da el _id del objeto insertado, indicando que la operación ha resultado en la creación de un nuevo documento.

En ocasiones nos interesará añadir una clave al documento asociado a un *upsert*, pero solo en el caso en el que se haya insertado el documento, es decir, solo si no existía con anterioridad. Esto se logra mediante el operador `$setOnInsert`.

En el caso del cliente que se da de baja, podemos suponer que todos los clientes tienen una clave `permanencia` con el número de meses que hace que iniciaron la relación con la empresa. En el caso de que al darse de baja no exista, puede interesarnos poner este valor a 0:

```
> db.clientes.updateOne({nombre:'Bertoldo'},{$set:{baja:true},
                    $setOnInsert:{permanencia:0}},{upsert:true})
{
   "acknowledged" : true,
   "matchedCount" : 0,
   "modifiedCount" : 0,
   "upsertedId" : ObjectId("5b3129f2ceb80d4e45f084ac")
}
```

Podemos comparar el resultado (asumiendo que Bertoldo no estaba y se ha producido una inserción):

```
> db.clientes.find()
{ "_id" : ObjectId("5b3129f2ceb80d4e45f084ac"),
  "nombre" : "Bertoldo", "baja" : true, "permanencia" : 0 }
```

Remove

Eliminar elementos en MongoDB es muy sencillo. Basta con que indiquemos un criterio de selección, y el sistema eliminará todos los documentos de la colección que deseemos:

```
> db.estelar.remove({clase:'planeta enano'})
WriteResult({ "nRemoved" : 2 })
```

Si solo deseamos eliminar un documento, el primero encontrado que verifique las condiciones, podemos añadir la opción `'justone'`:

```
> db.estelar.remove({clase:'planeta'},{justOne:true})
WriteResult({ "nRemoved" : 1 })
```

Si utilizamos como selección el documento vacío (`{}`) borraremos la colección completa. Podemos preguntarnos cuál es la diferencia —si la hay— con llamar a la función `drop()`. La diferencia es que `remove` eliminará los documentos uno a uno, de forma que al final tendremos una colección vacía, en la que, por ejemplo, los índices

asociados seguirán existiendo. En cambio, `drop()` eliminará por completo la colección y todos sus objetos asociados.

REFERENCIAS

- Documentación oficial de MongoDB (accedida en junio de 2018) `https://docs.mongodb.com/manual/`
- A. Sarasa. Introducción a las bases de datos NoSQL usando MongoDB. UOC, 2016.
- K.Chodorow. MongoDB: the definitive guide. O'Reilly. 2013.
- D. Hows, P.Membrey, y E.Plugge. *MongoDB Basics*. Apress. 2014.
- A. Nayak. *MongoDB Cookbook*. PacktPub, 2014.

5 APRENDIZAJE AUTOMÁTICO CON SCIKIT-LEARN

INTRODUCCIÓN

En los anteriores capítulos hemos tratado cómo recolectar datos desde distintas fuentes como ficheros, servicios web a través su API o analizando páginas web; y también hemos aprendido a almacenar esa gran cantidad de datos de manera eficaz. Sin embargo, los datos por si solos tienen poco valor y es necesario procesarlos para extraer información de ellos. En este capítulo nos centraremos en cómo realizar aprendizaje automático usando la biblioteca scikit-learn de Python para extraer patrones a partir de los datos. Concretamente, obtendremos modelos que nos permitirán predecir ciertos valores importantes a partir de otros, y modelos que nos servirán para distribuir individuos en grupos similares. Antes de entrar en profundidad con el aprendizaje automático en general y la biblioteca scikit-learn en particular presentaremos NumPy y Pandas, dos bibliotecas muy útiles para el análisis de datos en Python y que están fuertemente relacionadas con scikit-learn.

NUMPY

NumPy es una biblioteca Python que proporciona tipos de datos para almacenar de manera eficiente secuencias de valores numéricos y operar sobre ellos. NumPy almacena estos valores numéricos con un tamaño fijo y los almacena en regiones contiguas de memoria, lo que permite una comunicación sencilla con otros lenguajes de programación como C, C++ y Fortran (de hecho, varias funciones de la biblioteca están escritas en estos lenguajes para maximizar el rendimiento). Sobre estos datos,

NumPy permite realizar operaciones matemáticas del álgebra lineal o algoritmos más avanzados como la transformada de Fourier.

Desde el punto de vista de tipos de datos, NumPy ofrece `ndarray`, un array n-dimensional de elementos el mismo tipo. El acceso a estos elementos se realiza de manera natural mediante sus coordenadas en el espacio n-dimensional. NumPy ofrece distintos tipos de datos numéricos, entre los que destacan:

- `int8`, `int16`, `int32` e `int64`: enteros con signo de 8, 16, 32 y 64 bits.
- `uint8`, `uint16`, `uint32` e `uint64`: enteros sin signo de 8, 16, 32 y 64 bits.
- `float16`, `float32` y `float64`: números en coma flotante de 16, 32 y 64 bits.
- `complex64` y `complex128`: números complejos formados por dos números en coma flotante de 32 y 64 bits, respectivamente.

No vamos a entrar en detalle sobre cómo crear u operar con objetos `ndarray` ya que no los usaremos directamente en este libro, sino que son usados internamente por Pandas y scikit-learn y aparecerán de manera indirecta al realizar aprendizaje automático. No obstante, vamos a mostrar un pequeño ejemplo en el que se crea un vector de 3 dimensiones y una matriz 3x3 y se opera con ellos. Para ello cargaremos la biblioteca `numpy` con el nombre `np`, tal y como es usual, y la biblioteca de álgebra lineal:

```
>>> import numpy as np
>>> from numpy import linalg
```

A partir de `np` podremos crear objetos de tipos `ndarray` y acceder a sus elementos:

```
>>> v = np.array([1,2,3])
>>> print(v)
[1 2 3]
>>> print(v[1])
2
>>> m = np.array([[1,2,3],[0,1,4],[5,6,0]])
>>> print(m)
[[1 2 3]
[0 1 4]
[5 6 0]]
>>> print(m[0,0])
1
>>> print(m[2,1])
6
```


Con este código hemos usado la función `array` para crear un vector `v` con 3 elementos (objeto `ndarray` de una dimensión) y una matriz `m` de tamaño 3x3 (objeto `ndarray` de dos dimensiones). El acceso a sus elementos se realiza con el operador [], indicando los índices en cada dimensión. A continuación, realizamos algunas operaciones de álgebra lineal sobre ellos: calculamos su multiplicación y calculamos la inversa de la matriz:

```
>>> print(v @ m) # Multiplicación de vector y matriz
[16 22 11]
>>> m_inv = linalg.inv(m) # Inversa de la matriz 'm'
>>> print(m_inv)
[[-24.  18.   5.]
 [ 20. -15.  -4.]
 [ -5.   4.   1.]]
>>> print(m @ m_inv) # Multiplicacion de 'm' por su inversa
[[ 1.00000000e+00 -3.55271368e-15  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

Obsérvese que el resultado es la matriz identidad, aunque uno de los elementos no almacena exactamente un 0 sino un valor muy cercano: -3.55×10^{-15}.

PANDAS (*PYTHON DATA ANALYSIS LIBRARY*)

Pandas es una biblioteca Python muy utilizada para el análisis de datos. Está construida sobre NumPy, y proporciona clases muy útiles para analizar datos como `Series` o `DataFrame`. `Series` permite representar una secuencia de valores utilizando un índice personalizado (enteros, cadenas de texto, etc.) para acceder a ellos. Por otro lado, `DataFrame` nos permite representar datos como si de una tabla o una hoja de cálculo se tratase. Un objeto `DataFrame` dispone de varias columnas etiquetadas con cadenas de texto, y cada una de ellas está indexada. De esta manera podremos acceder fácilmente a cualquier celda a partir de sus coordenadas. No es nuestra intención mostrar todas las capacidades de Pandas, que son muchas, y únicamente presentaremos la clase `DataFrame` con las principales operaciones que nos pueden ayudar a cargar un conjunto de datos y procesarlo para realizar aprendizaje automático posteriormente. En el apartado de referencias indicamos algunos libros para profundizar en el uso de Pandas.

El conjunto de datos sobre los pasajeros del Titanic

A lo largo de este capítulo y de los siguientes vamos a utilizar el conjunto de datos sobre los pasajeros del Titanic, un conjunto de datos muy popular a la hora de practicar aprendizaje automático. Este conjunto está accesible desde `https://github.com/agconti/kaggle-titanic` con licencia Apache, aunque también lo hemos incluido en el repositorio del libro para comodidad del lector.

El conjunto de datos que vamos a considerar está almacenado en un fichero CSV de 891 filas y 12 columnas para cada fila:

1. **PassengerId:** identificador único de cada pasajero, números naturales consecutivos comenzando desde 0.
2. **Survived:** indica si el pasajero sobrevivió (valor 1) o pereció (valor 0).
3. **Pclass:** clase del billete comprado, que puede ser primera clase (1), segunda clase (2) o tercera clase (3).
4. **Name:** nombre completo del pasajero, incluyendo títulos como “Mr.”, “Mrs.”, “Master”, etc. Se representa como una cadena de texto.
5. **Sex:** sexo del pasajero, que puede ser “female” o “male”. Se representa como una cadena de texto.
6. **Age:** edad del pasajero como número real. En esta columna existen 177 filas que carecen de dicho valor.
7. **SibSp:** número de hermanos o cónyuges que viajaban en el Titanic. Cuenta también hermanastros, pero no amantes o personas comprometidas para casarse. Esta columna almacena un número natural.
8. **Parch:** número de padres e hijos del pasajero que viajaban en el Titanic. Tiene en cuenta hijastros. Algunos niños viajaban a cargo únicamente de su cuidador/a, así que en esos casos la columna tiene el valor 0. Esta columna almacena un número natural.
9. **Ticket:** número que identifica el billete adquirido. Se representa como una cadena de texto y toma 681 valores diferentes.
10. **Fare:** tarifa pagada al comprar el billete, representado como un número real positivo.
11. **Cabin:** número de camarote en el que se alojaba el pasajero, representado como una cadena de texto. Existen 148 valores diferentes así que había bastantes pasajeros que compartían camarote.
12. **Embarked:** puerto en el que embarcó el pasajero. Toma 3 valores representados como cadenas de texto: “C” para Cherbourg, “Q” para Queenstown y “S” para Southampton. Hay 2 filas a las que les falta este valor.

Como se puede observar, hay columnas que almacenan números enteros, números naturales y hasta cadenas de texto. Además, algunas columnas carecen de valores en algunas filas, lo que se conoce como valores vacíos (missing values). En esta sección veremos cómo cargar el conjunto de datos, extraer algunos datos para conocerlo mejor y finalmente transformarlo para que sea más fácil realizar aprendizaje automático sobre él.

Cargar un DataFrame desde fichero

Cargar un `DataFrame` a partir de un fichero es muy sencillo, ya que Pandas soporta un amplio catálogo de formatos: CSV, TSV, JSON, HTML, Parquet, HDF5… Como los datos sobre pasajeros del Titanic están almacenados en un fichero CSV utilizaremos la función `read_csv` de la biblioteca `pandas`. Esta biblioteca tradicionalmente se importa renombrándola a `pd`, costumbre que seguiremos en este libro:

```
>>> import pandas as pd
>>> df = pd.read_csv('data/titanic.csv')
```

La carga ha sido muy sencilla porque hemos utilizado los valores por defecto para todos sus parámetros. Sin embargo, Pandas nos permite seleccionar el carácter separador (se podría cambiar a `'\t'` para leer ficheros TSV), indicar manualmente el nombre de las columnas si el fichero no tiene cabecera, determinar diferentes valores que deben considerar como `True` y `False`, o seleccionar una codificación concreta del fichero. El número total de parámetros soportados excede de 50, así que recomendamos consultar la documentación de Pandas para configurar adecuadamente el proceso de lectura.

De la misma manera, Pandas proporciona las funciones `read_json`, `read_html`, `read_parquet`, etc. Uno de estos métodos, que ya mencionamos en el capítulo 1, es `read_excel` que nos permite cargar información desde hojas de cálculo de Microsoft Excel. Por ejemplo, cargar el fichero `subvenciones_totales.xls` generado en el capítulo 1 sería tan sencillo como:

```
>>> subvenciones =
pd.read_excel('data/Cap6/subvenciones_totales.xls',
sheet_name=None)
>>> subvenciones['Totales']
             Asociación  Importe total  Importe justificado  Restante
0  AMPA ANTONIO MACHADO        2344.99                    0  -2344.99
1    AMPA BACHILLER (…)        3200.00                    0  -3200.00
2         AMPA CASTILLA        2604.44                    0  -2604.44
(…)
```

Por defecto únicamente carga la primera página del fichero, por eso hemos incluido el parámetro `sheet_name=None` para que cargue todas las hojas del fichero y devuelva un diccionario ordenado de objetos `DataFrame`, asociados al nombre de la hoja. A partir de este diccionario podemos acceder a cualquier hoja a partir de su nombre, como en `subvenciones['Totales']`. En el caso de seleccionar una única hoja devolvería directamente un `DataFrame`. Por defecto utiliza los valores de la fila 0 como nombres de columna, aunque se puede cambiar a través del parámetro `header`. Al igual que `read_csv`, `read_excel` admite una veintena de parámetros para configurar de manera precisa cómo se lee el fichero y se vuelca en un `DataFrame`. Es importante darse cuenta de que las fórmulas no son incorporadas al `DataFrame`, sino que únicamente se incluye el valor calculado al abrir el fichero. De esta manera, si actualizamos el valor de la columna *Importe total* de la fila 0, el valor de la columna *Restantes* no se verá afectado, aunque en la hoja de cálculo original dicha celda tomaba el valor a partir de una fórmula.

Visualizar y extraer información

Una vez hemos cargado un objeto `DataFrame`, visualizar su contenido es tan sencillo como devolver dicho valor en una celda de Jupyter. El sistema mostrará una tabla interactiva en la que veremos marcada la fila actual según nos desplazamos.

```
>>> df
```

Si en lugar de usar las facilidades de Jupyter invocamos a la función `print`, entonces el `DataFrame` será representado como una cadena de texto y mostrado. Es muy posible que no quepan todas las columnas en pantalla, por lo que la visualización dividirá las columnas a mostrar y lo indicará con el carácter '\'. En cualquiera de las dos opciones, si el `DataFrame` es demasiado grande su salida será truncada, mostrando únicamente las primeras y últimas filas.

```
>>> print(df)
PassengerId  Survived  Pclass  \
0              1         0       3
1              2         1       1
2              3         1       3
(…)
```

Una de las primeras cosas que querremos saber sobre un `DataFrame` será su tamaño y el nombre de sus columnas. Esta información se puede obtener a partir de sus atributos `columns` y `shape`. `columns` nos devuelve un índice de Pandas, mientras que `shape` nos devuelve una pareja con el número de filas y el número de columnas.

En este caso vemos que estamos tratando con una tabla de 891x12, donde las columnas tienen los nombres que ya conocemos:

```
>>> df.columns
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age',
'SibSp','Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
dtype='object')
>>> df.shape
(891, 12)
```

Los `DataFrames` de Pandas admiten diversas formas de acceder a su contenido, aunque lo más común es utilizar los atributos accesores `iloc` y `loc`. Ambos sirven para acceder a una porción de la tabla, aunque varían en los parámetros que admiten. `iloc` recibe números indicando las posiciones de las filas y columnas deseadas. Si se pasa un único parámetro, se considera como el índice o índices de las filas a mostrar. Si se pasan dos parámetros, el primero será el índice o índices de las filas, y el segundo será el índice o índices de las columnas a seleccionar. Los parámetros pueden ser números, rangos o listas de números. Hay que tener cuidado, ya que los parámetros se pasarán utilizando corchetes `[]` en lugar de los paréntesis que son comunes en las invocaciones a métodos (porque realmente estamos invocando al método `__getitem__` del atributo `iloc` y no a un método directo del `DataFrame`). Por ejemplo:

- `df.iloc[5]` → Fila en la posición 5, es decir, la 6ª fila.
- `df.iloc[:2]` → Filas en el rango [0,2), la fila en posición 2 no es incluida.
- `df.iloc[0,0]` → Celda en la posición (0,0).
- `df.iloc[[0,10,12],3:6]` → Filas 0, 10 y 12; y de ellas las columnas con posiciones en el rango [3,6). La columna en posición 6 no es incluida.

Por otro lado, tenemos el atributo `loc` para referirnos a fragmentos de la tabla utilizando los índices. En el índice horizontal tendremos los nombres de las columnas, y en el índice vertical usualmente tendremos posiciones empezando desde 0. Esta manera de acceder a los contenidos es más cómoda, ya que podemos utilizar los nombres y por tanto no tendremos que contar las posiciones previamente. `loc` recibe uno o dos parámetros, con el mismo funcionamiento que en `iloc`. Algunos ejemplos del uso de `loc` serían:

- `df.loc[0]` → Fila con índice 0.
- `df.loc[0,'Fare']` → Fila con índice 0 y columna *Fare*.

- `df.loc[:3, 'Sex':'Fare']` → Filas con índices en el rango [0,3] y columnas entre *Sex* y *Fare*. En este caso ambos extremos se incluyen, tanto en filas como en columnas.
- `df.loc[:3, ['Sex','Fare','Embarked']]` → Filas con índices en el rango [0,3] y columnas con nombre *Sex*, *Fare* y *Embarked*.

Ahora conocemos las columnas que existen y hemos podido consultar algunas filas y celdas. Sin embargo, para conocer mejor los datos que estamos tratando necesitamos conocer los tipos de datos concretos que almacena cada columna. Para ello únicamente debemos acceder al atributo `dtypes`, que es un objeto `Series` de Pandas indexado por nombre de columna. Los tipos de datos que mostrará son los mismos que vimos para NumPy, donde `object` usualmente significará que se trata de una cadena de texto. Para nuestro `DataFrame df` con los datos de los pasajeros del Titanic vemos que las columnas *PassengerId*, *Survived*, *Pclass*, *SibSp* y *Parch* están representados como enteros de 64 bits, las columnas *Age* y *Fare* como números en coma flotante de 64 bits, mientras que las columnas *Name*, *Sex*, *Ticket*, *Cabin* y *Embarked* almacenan cadenas de texto.

```
>>> df.dtypes
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

Para las columnas numéricas, Pandas nos puede proporcionar una descripción completa de los valores almacenados gracias al método `describe`, que nos devuelve un nuevo `DataFrame` resumen. Esto nos permite conocer rápidamente algunas medidas estadísticas como la media, la desviación típica, valores mínimos y máximos e incluso los cuartiles. También nos indica el número de valores incluidos, lo que nos permite saber la cantidad de valores vacíos que existen en cada columna. La salida para el `DataFrame` con los pasajeros del Titanic sería la siguiente, donde hemos omitido las columnas *Parch* y *Fare*:

```
>>> df.describe()
       PassengerId    Survived      Pclass         Age       SibSp
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000
```

Como se puede ver, la columna *Age* tiene 714 valores no vacíos, siendo 0,42 el valor mínimo y 80 el valor máximo. Además, sabemos que la edad media es de 29,7 años, con una desviación típica de 14,52. El método `describe` también nos permite conocer que la edad mediana es 28 años, y que los pasajeros en la mitad central tenían una edad entre 20,125 y 38 años (lo que se conoce como rango intercuartílico).

El método `describe` puede devolver medidas informativas también para las columnas no numéricas, aunque muchas de estas filas como la media o los cuartiles aparecerán como `NaN` ya que no se pueden calcular para este tipo de columnas (no tiene sentido calcular la media de dos cadenas de texto). Para que `describe` muestre todas las columnas debemos pasar el parámetro `include='all'`. El resultado sería el siguiente, donde hemos omitido todas las columnas numéricas:

```
>>> df.describe(include='all')
                           Name   Sex  Ticket  Cabin Embarked
count                       891   891     891    204      889
unique                      891     2     681    147        3
top     Larsson, Mr. Bengt Edvin  male    1601     G6        S
freq                          1   577       7      4      644
mean                        NaN   NaN     NaN    NaN      NaN
std                         NaN   NaN     NaN    NaN      NaN
min                         NaN   NaN     NaN    NaN      NaN
25%                         NaN   NaN     NaN    NaN      NaN
50%                         NaN   NaN     NaN    NaN      NaN
75%                         NaN   NaN     NaN    NaN      NaN
max                         NaN   NaN     NaN    NaN      NaN
```

Como se ve, al incluir columnas no numéricas han aparecido nuevas filas. Una muy interesante es `unique`, que nos indica el número de valores diferentes que hay. Por ejemplo, tenemos 2 valores diferentes para la columna *Sex*, 3 valores para *Embarked* y 891 para *Name* (es decir, no hay nombres repetidos). También aparecen

las filas `top`, que contienen el elemento más repetido, y `freq`, que indican el número de repeticiones de dicho elemento más común.

Transformar DataFrames

A continuación, vamos a presentar algunas de las transformaciones que se pueden realizar sobre un `DataFrame`. Nos vamos a centrar únicamente en aquellas que nos interesan para poder realizar posteriormente aprendizaje automático, aunque remitimos al lector a la documentación de Pandas o a los libros incluidos en la sección de referencias para profundizar en las transformaciones disponibles.

La primera transformación que vamos a realizar es eliminar algunas columnas que no nos parecen muy relevantes, concretamente *PassengerId*, *Name*, *Ticket* y *Cabin*. Para eliminar columnas de un DataFrame únicamente debemos invocar al método `drop` y pasar una lista de nombres en su parámetro `columns`:

```
>>> df = df.drop(columns=['PassengerId', 'Name', 'Ticket','Cabin'])
```

El siguiente paso será eliminar todas aquellas filas que tengan algún valor vacío. Para ello utilizaremos el método `dropna` con los parámetros por defecto, que eliminará aquellas filas con al menos un valor vacío:

```
>>> df = df.dropna()
```

El método `dropna` permite configurar cómo se decidirá si una fila se elimina o no, por ejemplo, indicando que haya un mínimo de valores vacíos (parámetro `thresh`) o requiriendo que todos los valores sean vacíos (parámetro `how`).

Por último, queremos transformar las columnas que almacenan cadenas de texto para que representen esa información como números naturales consecutivos a partir de 0. Concretamente queremos que la columna *Sex* (`'female'` o `'male'`) tome valores 0 y 1; y la columna *Embarked* (`'C'`, `'Q'` y `'S'`) tome valores 0, 1 y 2. Para ello vamos a reemplazar dichas columnas completamente usando el operador de selección `[]`. Este operador recibe el nombre de una de las columnas y la devuelve como una secuencia de valores indexados, concretamente un objeto de la clase `Series`. Estos objetos se pueden operar a través de sus métodos, y volver a introducir en el `DataFrame` original, por ejemplo:

```
>>> df['Sex'] = df['Sex'].astype('category').cat.codes
>>> df['Embarked'] = df['Embarked'].astype('category').cat.codes
```


En estas dos instrucciones seleccionamos una columna (`df['Sex']` y `df['Embarked']`) y mediante asignación la sustituimos por otros valores. Para calcular los valores numéricos seleccionamos la columna, la reinterpretamos como una categoría (`astype('category')`) y finalmente de esa categoría nos quedamos con la secuencia de su representación numérica (`cat.codes`). Al reinterpretar la columna como una categoría, se recorren los valores detectando los valores únicos y dándoles una representación numérica única. El operador `[]` nos devuelve un objeto de tipo `Series`, que nos permite operar sobre él (como reinterpretarlo como categoría) pero también asignarlo a otro objeto del mismo tipo, como hacemos aquí. De la misma manera podríamos añadir columnas mediante asignación utilizando el operador `[]` con un nombre nuevo de columna:

```
>>> df['Sex_num'] = df['Sex'].astype('category').cat.codes
```

En este caso la columna *Sex* permanecería igual, pero habríamos añadido una nueva columna con nombre *Sex_num*. Como hemos comentado, existen muchas operaciones que se pueden realizar sobre columnas. Por ejemplo, podríamos incrementar en uno la edad de todos los pasajeros (`df['Age'] + 1`), representar la edad en meses (`df['Age'] * 12`), obtener la secuencia booleana de pasajeros de edad avanzada (`df['Age'] > 70`), etc. Recomendamos a los lectores acudir al apartado de referencias para ahondar más en las amplias capacidades de Pandas a este respecto.

Salvar a ficheros

Una vez hemos realizado todas las transformaciones a nuestro conjunto de datos, el último paso será volcarlo a disco para poder reutilizarlo las veces que necesitemos. En este apartado Pandas ofrece las mismas facilidades que para su lectura, proporcionando diversos métodos para crear ficheros según el formato deseado. Por ejemplo, para volcar el `DataFrame` a un fichero CSV invocaríamos a `to_csv`:

```
>>> df.to_csv('data/Cap6/titanic_ml.csv', index=False)
```

Únicamente hemos necesitado indicar la ruta y además hemos añadido el parámetro `index=False` para que no incluya una columna inicial con el índice de cada fila (serían números naturales consecutivos comenzando en 0). `to_csv` admite más parámetros para configurar el carácter separador (`sep`), elegir qué columnas escribir (`columns`), seleccionar compresión (`compression`), etc.

Salvar un `DataFrame` en formato Excel (tanto XLS como XLSX) es igual de sencillo, pero usando el método `to_excel`:

```
>>> df.to_excel('data/Cap6/titanic_ml.xls',  index=False)
>>> df.to_excel('data/Cap6/titanic_ml.xlsx', index=False)
```

De la misma manera, `to_excel` admite diversos parámetros adicionales para configurar el proceso de creación del fichero, aunque en nuestros ejemplos hemos tomado los valores por defecto y únicamente hemos indicado `index=False` para que no se cree una columna inicial con los índices de cada fila.

Al invocar al método `to_excel` con la ruta de un fichero existente todo su contenido se perderá y únicamente contendrá la página creada a partir del `DataFrame`. Si queremos añadir hojas a un fichero existente, deberemos utilizar un objeto `ExcelWriter` en lugar de una ruta. Este objeto se crea directamente a partir de la ruta:

```
>>> writer = pd.ExcelWriter('data/Cap6/titanic_2.xlsx')
>>> df.to_excel(writer, sheet_name='Hoja1', index=False)
>>> df.to_excel(writer, sheet_name='Hoja2', index=False)
>>> writer.close()
```

En este caso hemos creado un fichero `titanic_2.xlsx` con las hojas `Hoja1` y `Hoja2`, que en este caso contendrán los mismos datos. Es importante invocar al método `close` del objeto `ExcelWriter` para garantizar que los datos son volcados al disco y el fichero se cierra convenientemente.

APRENDIZAJE AUTOMÁTICO

El *aprendizaje automático* (*machine learning* en inglés) es una rama de la informática que utiliza técnicas matemáticas y estadísticas para desarrollar sistemas que *aprenden* a partir de un conjunto de datos. Aunque el término *aprendizaje* es muy amplio, en este contexto consideramos principalmente la detección y extracción de patrones que se observan en el conjunto de datos usado para el aprendizaje. Ejemplos de estos patrones pueden ser detectar que es altamente probable sobrevivir al hundimiento del Titanic si se viajó en 1ª clase, descubrir que la mayoría de los clientes que compran leche también compran galletas el mismo día, o considerar que un usuario de una red social es muy similar a otros usuarios catalogados como "víctimas fáciles de noticias falsas". Los datos originales no son más que largos listados con información de los pasajeros de un barco, compras realizadas a lo largo del año en todos los establecimientos de una cadena de

alimentación, o datos de usuarios de una red social (por ejemplo, sus amigos, el texto de los mensajes escritos o la hora y localización de sus conexiones a la red social). Los datos por sí solos no nos dicen nada, pero al *aprender* sobre ellos extraemos *conocimiento* que nos proporciona un entendimiento más preciso de la realidad y nos permite tomar mejores decisiones. Este conocimiento nos puede ayudar a elegir el billete para viajar en un barco que cruce el océano Atlántico, elegir la distribución de los productos en nuestras tiendas o realizar una campaña de publicidad dirigida a ciertos usuarios de una red social utilizando noticias de dudosa calidad.

El aprendizaje automático existe desde hace varias décadas, pero es en los últimos años cuando ha recibido un interés especial, llegando a aparecer un perfil profesional específico conocido como *científico/a de datos*. Las razones de este auge son el aumento de la capacidad de almacenamiento y cómputo gracias a la nube, unido al incremento de los datos disponibles a partir de sensores (internet de las cosas), dispositivos móviles y redes sociales.

Nomenclatura

El aprendizaje automático toma como punto de partida un conjunto de datos. Este conjunto de datos está formado por varias *instancias*, cada una de ellas contiene una serie de *atributos*. Si pensamos en nuestro conjunto de datos como una tabla, cada fila será una instancia y cada columna será un atributo. Todas las instancias tienen el mismo número de atributos, aunque algunos de ellos pueden contener valores vacíos.

A su vez, cada atributo del conjunto almacenará valores de un tipo concreto. Por un lado, tenemos los *atributos categóricos*, que únicamente pueden tomar un conjunto prefijado de valores. Los atributos categóricos se dividen a su vez en dos grupos:

- *Atributos nominales*, donde los valores no tienen ninguna noción de orden o lejanía entre ellos. Un ejemplo puede ser el deporte favorito de una persona, ya que “fútbol” no es mayor que “rugby”, ni tenemos una manera de decidir si “fútbol” y “rugby” están más o menos lejos que “golf” y “tenis”. En el caso de los pasajeros del Titanic, el atributo *Embarked* que indica el puerto de embarque sería un atributo nominal ya que no tenemos una noción de orden entre distintos puertos.
- *Atributos ordinales*, donde los valores sí tienen un orden y noción de lejanía. En el caso de los pasajeros del Titanic, el atributo *Pclass* que indica la clase del billete es de tipo categórico ordinal porque toma únicamente 3 valores (1ª, 2ª

y 3ª) y además tenemos una noción de orden o lejanía (1ª es mejor que 3ª, y además la distancia entre 1ª y 3ª es mayor que la distancia entre 2ª y 3ª).

Por otro lado, también tenemos *atributos continuos* que almacenan valores numéricos arbitrarios en un rango dado. El conjunto de datos sobre pasajeros del Titanic contiene varios atributos continuos como la edad (*Age*), el precio del billete (*Fare*) o el número de hermanos y cónyuges (*SibSp*). Los atributos continuos pueden almacenar valores reales, pero también valores naturales o enteros.

Dependiendo de nuestras necesidades, dentro del conjunto de datos puede existir un atributo especial llamado *clase* que sirve para catalogar la instancia. Este atributo es importante, ya que normalmente determina el tipo de aprendizaje automático que queremos realizar.

Tipos de aprendizaje

Existen distintas maneras de catalogar los tipos de aprendizaje automático que se pueden aplicar. Por ejemplo, se puede distinguir entre *aprendizaje por lotes*, donde el aprendizaje se realiza una única vez usando todos los datos; o *aprendizaje en línea*, donde el aprendizaje se realiza poco a poco cada vez que aparecen nuevas instancias. También se puede distinguir entre un aprendizaje cuyo resultado es un modelo *caja negra* que nos sirve únicamente para catalogar nuevas instancias pero no sabemos cómo lo hace, o un aprendizaje cuyo resultado es un modelo que además de catalogar nuevas instancias nos permite también inspeccionar sus atributos internos y conocer en qué se basa para tomar las decisiones. Sin embargo, lo más usual es diferenciar los tipos de aprendizaje automático en relación con si existe un atributo clase y qué tipo tiene.

APRENDIZAJE SUPERVISADO

El *aprendizaje supervisado* se realiza sobre conjuntos de datos que tienen un atributo *clase*. En este tipo de aprendizaje se persigue ser capaz de predecir la clase a partir de los valores del resto de atributos. Dependiendo del tipo que tenga la clase se pueden distinguir a su vez dos familias:

- *Clasificación*, cuando la clase es un atributo categórico. Como la clase únicamente puede tomar un número finito de valores distintos, lo que queremos es clasificar una instancia en una de estas categorías diferentes. Un ejemplo de este tipo de aprendizaje automático sería predecir si un pasajero del Titanic sobrevive o no dependiendo del valor del resto de atributos. En este caso la clase sería el atributo *Survived* que toma valores 1 (sobrevivió) o 0

(no sobrevivió). Se trataría por tanto de *clasificación binaria*, en contraposición a la *clasificación multiclase* que se da cuando la clase puede tener más de 2 valores posibles. No todos los algoritmos de clasificación están diseñados para tratar más de dos clases, aunque usualmente se pueden extender a este tipo de clasificación utilizando técnicas como *one-vs-rest* o *one-vs-all* (ver más detalles en los libros incluidos en las referencias).

- *Regresión*, cuando la clase es un atributo continuo. En este caso no queremos clasificar una instancia dentro de un número predeterminado de categorías sino ser capaces de predecir un valor continuo a partir del resto de atributos. Un ejemplo de regresión sería tratar de predecir el precio de billete pagado por cada pasajero del Titanic, donde la clase sería el atributo *Fare*.

APRENDIZAJE NO SUPERVISADO

El aprendizaje no supervisado se da cuando no disponemos de ningún atributo *clase* que guíe nuestro aprendizaje. En estas ocasiones queremos encontrar patrones que aparezcan en nuestro conjunto de datos, sin tener ninguna guía o supervisión sobre lo que queremos encontrar. Un ejemplo claro es el análisis de grupos (*clustering*), que persigue dividir las instancias en conjuntos de elementos similares. Esto nos permitiría por ejemplo segregar usuarios en relación con su similitud, y posiblemente usar esa información para recomendar productos que otros usuarios similares han encontrado interesantes. Otro ejemplo clásico es poder asociar eventos que ocurren a la vez. Si se aplica este tipo de aprendizaje a las compras en un supermercado, podemos encontrar qué productos aparecen juntos en las compras de los clientes. Esta información nos puede permitir colocar estos productos en estanterías cercanas para acelerar las compras de los clientes, o alejarlos para obligar a los clientes a recorrer varios pasillos y fomentar así que compren más.

Proceso de aprendizaje y evaluación de modelos

A la hora de aplicar aprendizaje automático para obtener un modelo partimos de un conjunto de datos inicial. Si nos centramos en el aprendizaje supervisado, lo usual es dividir este conjunto en dos fragmentos separados: el *conjunto de entrenamiento* y el *conjunto de test*. El conjunto de entrenamiento nos servirá para entrenar nuestro algoritmo y obtener el modelo, mientras que el conjunto de test nos servirá para medir la calidad predictiva del modelo generado. Es importante que el conjunto de entrenamiento y el de test no compartan instancias, porque lo que queremos medir es lo bien que el conocimiento obtenido durante el entrenamiento se aplica a instancias nuevas no observadas anteriormente. Si utilizásemos las mismas instancias para crear el modelo y medir su calidad, realmente estaríamos midiendo lo bien que recuerda lo aprendido. En casos así estaríamos considerando como excelentes

modelos que han aprendido de memoria todas las instancias vistas, pero que a la hora de tratar con instancias nuevas tienen un pobre resultado porque no han sabido generalizar su conocimiento. Un modelo así se diría que está *sobreajustado*, y en la práctica nos serviría de poco. Para evitar este tipo de situaciones se suele realizar un muestreo aleatorio seleccionando un 70%-80% de instancias para el entrenamiento y el resto de instancias para la evaluación de la calidad. En clasificación normalmente es interesante asegurar que haya suficientes ejemplares de cada clase en estos dos conjuntos, así que se suele aplicar un *muestreo estratificado* por cada valor de clase. La figura 5-1 muestra una representación gráfica de las distintas etapas involucradas en aprendizaje supervisado (clasificación o regresión).

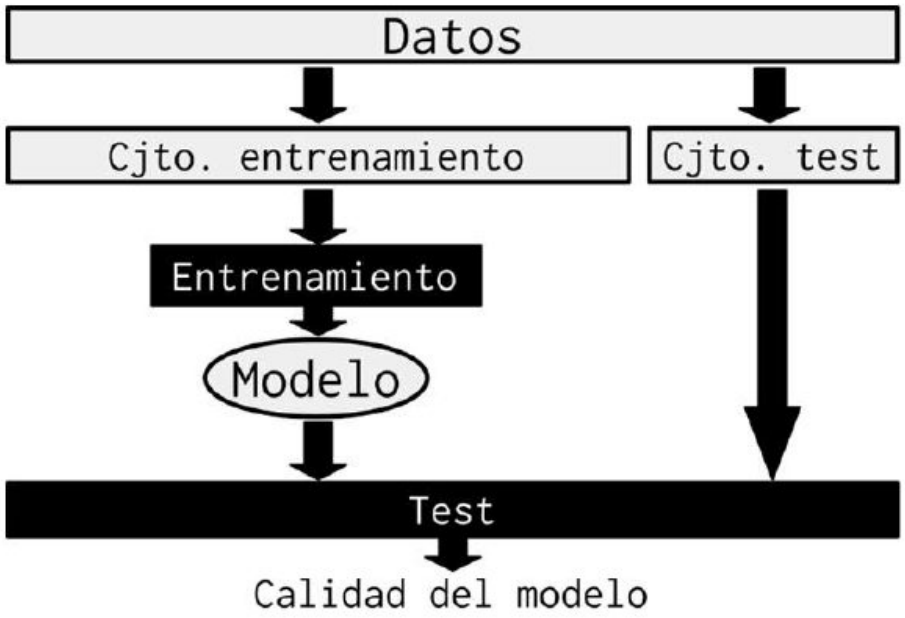


Figura 5-1. Proceso de aprendizaje supervisado.

Una vez tenemos el conjunto de entrenamiento, el siguiente paso es entrenar el algoritmo para que produzca un modelo. Suele ocurrir que el algoritmo elegido aceptará distintos hiperparámetros que pueden afectar a la calidad del modelo final, por ejemplo, el número de iteraciones a aplicar, el valor de algunos umbrales, etc. ¿Qué parámetros son los adecuados para mi conjunto de datos concreto? Esta pregunta es difícil de contestar *a priori*, y normalmente se entrena el mismo algoritmo con distintos hiperparámetros para detectar aquellos que generan el mejor modelo. Para evaluar esta calidad "intermedia" no queremos utilizar el conjunto de test, puesto que lo hemos separado para que nos dé una medida de la calidad de nuestro modelo final y no para tomar decisiones. Por lo tanto, todas las decisiones que tengamos que tomar deberán considerar únicamente el conjunto de entrenamiento. Para ello se aplica validación cruzada (*cross validation*). En el caso

más sencillo se separa un *conjunto de validación* del conjunto de entrenamiento, se obtienen los modelos con los distintos valores de los hiperparámetros, y se elige el modelo que mejores resultados obtiene sobre el conjunto de validación. Esto tiene la desventaja de que hemos separado algunas instancias del aprendizaje para realizar la validación, y en situaciones en las que no disponemos de muchas instancias eso puede resultar en modelos de menor calidad. Para solucionar este problema se pueden aplicar técnicas de validación cruzada más avanzadas como la *validación cruzada de K iteraciones* (*k-fold cross-validation*). La figura 5-2 muestra el proceso de aprendizaje supervisado utilizando un conjunto de validación. Como se puede ver se entrena el mismo algoritmo con distintos hiperparámetros *p1* y *p2* y finalmente se utiliza el conjunto de validación para elegir el mejor modelo.

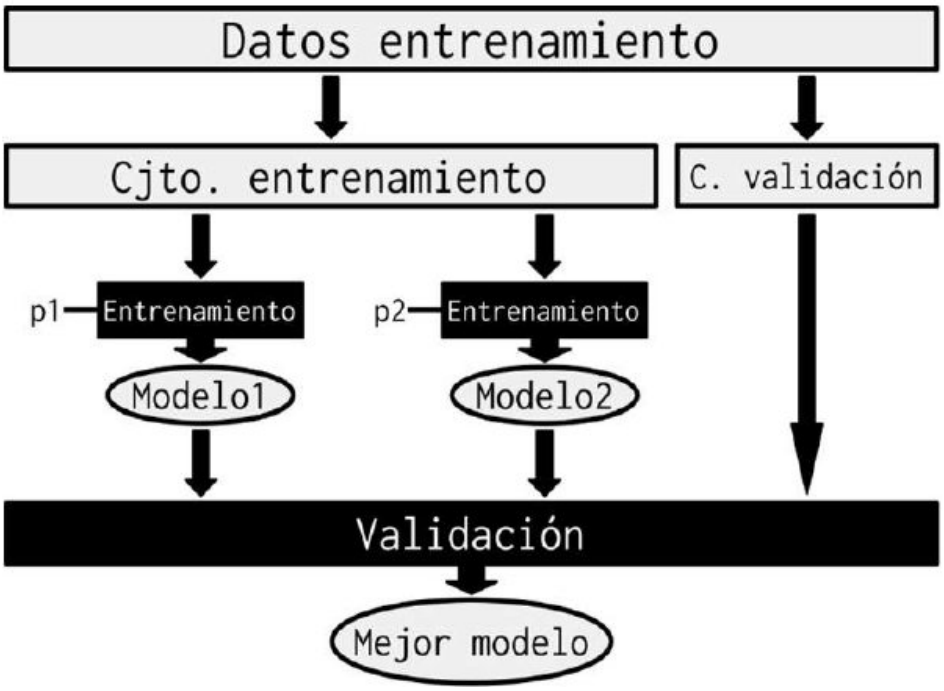


Figura 5-2. Entrenamiento supervisado usando un conjunto de validación.

Independientemente de si hemos probado distintos valores para los hiperparámetros o no, el resultado de la etapa de entrenamiento será un modelo. Este modelo nos servirá para predecir la clase, pero antes de sacarlo a producción querremos obtener una medida de su calidad a partir del conjunto de test que separamos previamente. Existen múltiples métricas que se pueden utilizar para esta tarea, que varían dependiendo del tipo de aprendizaje aplicado. Para clasificación binaria podemos destacar la *exactitud* (*accuracy*), también llamada tasa de aciertos, que mide la proporción de instancias correctamente clasificadas entre el total de instancias probadas. En el caso de regresión existen varias métricas disponibles que tratan de medir la cercanía entre las predicciones del modelo y la realidad. Entre ellas

podemos destacar el error *cuadrático medio* (*mean squared error* o MSE) o el error absoluto medio (*mean absolute error*, MAE), definidos como:

$$MSE(y, y^*) = \frac{1}{n}\sum_{i=1}^{n-1}(y_i - y_i^*)^2 \qquad MAE(y, y^*) = \frac{1}{n}\sum_{i=1}^{n-1}|y_i - y_i^*|$$

donde consideramos un conjunto de test con *n* elementos, y donde los valores y_i e y_i^* son la clase real y predicha para la i-ésima instancia, respectivamente. Como se puede ver a partir de la fórmula, en el cálculo del MSE las diferencias se elevan al cuadrado, por lo que obtiene resultados que están en otro orden de magnitud con respecto a los valores originales de la clase. Por ello, en algunos casos se utiliza la raíz cuadrada del MSE, llamado *root mean squared error* o RMSE.

Aplicar aprendizaje no supervisado es similar a aplicar clasificación o regresión con la excepción de que no debemos separar ningún conjunto de test, por tanto, todos los datos de los que dispongamos los usaremos para el entrenamiento. Al igual que en el aprendizaje supervisado, los algoritmos pueden admitir distintos hiperparámetros, así que aplicaremos validación cruzada para quedarnos con el mejor modelo generado. Sin embargo, en este tipo de aprendizaje no tenemos un atributo *clase* que nos indica si la predicción ha sido adecuada o no, por lo que evaluar la calidad del modelo es más complicado.

Para el análisis de grupos podemos calcular una medida numérica indicando lo *cohesionados* que están los clústeres encontrados. Idealmente preferiremos clústeres más cohesionados donde los elementos se parezcan más entre sí que al resto de clústeres. Métricas que siguen esta intuición son el *coeficiente de silueta* (*Silhouette coefficent* o *score*) o el *índice Calinski-Harabasz*. En algunas ocasiones podemos disponer de información externa que nos aproxima los distintos clústeres que hay en nuestro conjunto de datos. En esas ocasiones podemos medir la calidad del agrupamiento generado frente a la información externa que tomamos como fidedigna.

Con respecto a la inferencia de reglas de asociación, también existen distintas métricas de calidad que podemos utilizar. Este tipo de reglas tiene una condición y un resultado, por ejemplo "*SI leche Y galletas ENTONCES yogures*". Generalmente preferiremos reglas *confiables* que sean válidas en una cantidad amplia de instancias, es decir, que si una instancia cumple su condición entonces verifique su resultado con una alta probabilidad. También preferiremos reglas de amplio *soporte* cuya condición y cuyo resultado se verifiquen en muchas instancias, lo que indicará que se trata de un conocimiento muy generalizado. Usando estas y otras métricas podremos

comparar los distintos modelos de reglas de asociación generados y elegir el que consideramos mejor durante la fase de validación cruzada.

Etapa de preprocesado

Como hemos visto en el apartado anterior, el aprendizaje automático se realiza y se evalúa utilizando conjuntos de instancias. Todas las instancias tendrán el mismo número de atributos, y cada atributo almacenará valores del mismo tipo. Independientemente de dónde procedan los datos (ficheros, servicio web a través de un API, *web scrapping*, etc.), lo más usual es que los hayamos consolidado en un único `DataFrame` de la biblioteca `pandas`. Pero antes de comenzar con el aprendizaje es importante *preprocesar* este conjunto para obtener datos de la mejor calidad posible.

En nuestro `DataFrame` tendremos distintas columnas, pero es muy posible que no queramos utilizar todas ellas para realizar el aprendizaje automático. Por ello el primer paso a realizar es la *selección* de los atributos que nos interesan. Por ejemplo, el fichero con los datos de los pasajeros del Titanic contiene 11 atributos más la clase, pero no todos parecen demasiado útiles. En este grupo podríamos destacar el identificador de pasajero, que es un número único para cada persona, o el nombre de cada pasajero. ¿Acaso llamarse John puede aumentar la probabilidad de sobrevivir a un naufragio?

En algunas ocasiones nos podemos encontrar con atributos que no parecen muy relevantes, pero de los que quizá se podría *extraer* información interesante. El camarote donde viajaba un pasajero es una cadena de texto con un identificador alfanumérico, y como tal no nos aporta mucha información. Además, cada camarote alojaba un número reducido de pasajeros, pero lo que habrá muchos valores diferentes. Sin embargo, podríamos procesar esa información y extraer "en qué cubierta estaba el camarote", "cuánta distancia había hasta el bote de emergencia más cercano" o "cuánta gente se alojaba en ese camarote". Esta información, que parece más relevante que el nombre del camarote, no estaba en el conjunto de datos original, pero se ha podido extraer de él (quizá utilizando información adicional). Otro ejemplo sería el nombre, que en principio nos ha resultado totalmente irrelevante. Sin embargo, podría venir acompañados con títulos como *Master*, *Doctor*, *Sir*, *Lord*, etc. Podría ser interesante almacenar para cada pasajero si tiene algún título, o incluso distinguir entre los distintos títulos que aparecen, dado que puede ser un *proxy* del nivel socioeconómico del pasajero.

Otro paso importante durante el preprocesado es decidir qué hacer con los valores vacíos. La opción más sencilla y más drástica es eliminar completamente todas aquellas instancias que contengan algún valor vacío. Sin embargo, esto puede hacer que en algunas situaciones perdamos información valiosa. Por ello, otra opción sería asignar algún valor concreto a estos valores vacíos, usando por ejemplo el valor promedio del atributo, el valor máximo o la moda. Esta transformación es estándar en casi todos los sistemas de aprendizaje automático y se conoce como *imputer*.

Como veremos más adelante, las bibliotecas de aprendizaje automático están diseñados para aceptar únicamente atributos con valores numéricos (enteros o reales). Algunos algoritmos no tendrían ningún problema en aceptar atributos categóricos que almacenan cadenas de texto, pero se toma esta decisión para proporcionar una interfaz homogénea a todos los algoritmos de la biblioteca. Por ello, es necesario traducir todo lo que no sea numérico a números, y la opción más usual es asignar a cada valor diferente un número natural consecutivo empezando desde 0. Es importante darse cuenta de que esto no cambiará la *naturaleza* de un atributo, únicamente la representación de sus valores. De esta manera, el atributo *Embarked* del conjunto sobre pasajeros del Titanic seguirá siendo categórico nominal si cambiamos las cadenas `'C'`, `'Q'` y `'S'` por los valores numéricos 0, 1 y 2, respectivamente. Sin embargo, hay que tener cuidado al representar estos valores, porque lo que interpretará un algoritmo de aprendizaje automático con esta nueva representación es que `'C'` y `'S'` están *más lejos* que `'Q'` y `'S'` (0--2 frente a 1--2). Esta información errónea no aparecía en nuestro conjunto original, sino que la hemos introducido nosotros durante el preprocesado. Para evitar esta situación indeseada, los atributos categóricos nominales (los que no tienen noción de orden ni lejanía) se suelen codificar utilizando la técnica conocida como *one hot encoding*. En ella, un atributo nominal de *n* valores generará *n* atributos nominales binarios *atributo_i* que contendrán un 1 si la instancia contenía el valor i-ésimo en dicho atributo, y 0 en otro caso. Por ejemplo, el atributo *Embarked* daría lugar a 3 atributos que podíamos llamar *Embarked_C*, *Embarked_Q* y *Embarked_S*. Si una instancia tenía originalmente el valor *Embarked=Q* entonces tras la transformación tendrá los valores *Embarked_C=0*, *Embarked_Q=1* y *Embarked_S=0*. La siguiente 6-3 ilustra esta transformación para los 3 posibles valores de *Embarked*. Gracias a *one hot encoding* conseguimos que las instancias que tienen el mismo valor de *Embarked* tengan exactamente los mismos valores en los atributos transformados (lejanía mínima), mientras que instancias con valores diferentes diferirán siempre 2 atributos transformados (lejanía máxima). Así que hemos conseguido codificar las nociones de "ser igual" y "ser distinto" sin imponer un orden espurio.

Embarked		Embarked_C	Embarked_Q	Embarked_S
C	→	1	0	0
Q	→	0	1	1
S	→	0	0	1

Figura 5-3. Codificacion one hot enconding para el atributo Embarked.

La técnica de *one hot encoding* se puede aplicar también a los atributos categóricos ordinales, aunque en este caso se podría utilizar el propio orden de los valores. Por ejemplo, si tenemos un atributo categórico nominal con los valores "Muy poco", "poco", "normal", "a menudo" y "muy a menudo" podemos realizar la codificación "Muy poco"=0, "poco"=1, "normal"=2, "a menudo"=3 y "muy a menudo"=4. Esta codificación respeta el orden y también la noción intuitiva de lejanía: "poco" está igual de lejos de "normal" que de "muy poco". En este caso la codificación encaja muy bien porque los valores están uniformemente separados, aunque se podrían establecer diferencias superiores a 1 en aquellos valores que estén más lejanos. En todo caso, si existe alguna duda sobre la codificación utilizada para un atributo nominal la recomendación general es utilizar *one hot encoding*.

Tras estos pasos tenemos un `DataFrame` formado por atributos útiles y todos los valores están representado como números, pero aún no hemos terminado: nos falta uniformizar los rangos de los atributos. En el ejemplo de los pasajeros del Titanic tenemos dos atributos continuos como son *Age* (edad en años) y *SibSp* (número de hermanos y cónyuges). El atributo *Age* toma valores entre 0 y 80, mientras que *SibSp* toma valores entre 0 y 8. A la hora de extraer patrones, un algoritmo de aprendizaje automático considerará que la diferencia entre 0 años y 80 años es muy superior a la diferencia entre 0 hermanos/cónyuges y 8 hermanos/cónyuges. Y realmente no es el caso, puesto que se trata de la diferencia máxima en cada uno de los atributos. Además, en casos como *Age* el valor concreto (y su diferencia) depende de una decisión arbitraria como es la unidad de medida elegida. ¿Acaso la diferencia entre 0 años y 80 años es distinta de la diferencia entre 0 siglos y 0,8 siglos? Para solucionar este problema se recomienda realizar una fase de *escalado* de atributos. Existen distintas posibilidades: escalar cada atributo para que esté en el rango [0, 1] (*MinMax scaler*), escalar para que esté en el rango [-1, 1] (*MaxAbs scaler*), o escalar para medir el número de desviaciones típicas que un valor se aleja de su media, es decir, cambiarlo por su *z-score* (*Standard scaler*). Distintos algoritmos de aprendizaje automático hacen distintas suposiciones sobre la distribución de los atributos que

aceptan, así que la elección de un escalado u otro depende mucho de la técnica que vayamos a aplicar.

BIBLIOTECA SCIKIT-LEARN

Podríamos decir que scikit-learn es la biblioteca de aprendizaje automático más popular para Python. Proporciona una gran cantidad de algoritmos de aprendizaje automático (clasificación, regresión y análisis de grupos) además de distintas técnicas de preprocesado y de evaluación de modelos. Además, todas las clases de scikit-learn proporcionan una interfaz común muy sencilla, lo que facilita mucho su aprendizaje y utilización. Otro de los puntos destacados de scikit-learn es que encaja muy bien con Numpy y Pandas, permitiendo utilizar fácilmente los datos que disponemos sin requerir cambios de formatos. Por último, scikit-learn es un proyecto de código abierto con una comunidad bastante activa y una documentación muy extensa. Por todo ello scikit-learn es la primera opción a la que se suele acudir cuando necesitamos analizar datos en Python.

En esta sección vamos a presentar los principios básicos del uso de scikit-learn a la hora de realizar aprendizaje automático sobre los datos de los pasajeros del Titanic. Nos centraremos en aplicar una serie de etapas de preprocesado y realizar un ejemplo de clasificación, otro de regresión y otro de análisis de grupos. Sin embargo, animamos al lector a consultar la documentación de scikit-learn para conocer el resto de técnicas que tiene disponibles. En todos los ejemplos de este apartado usaremos la versión de scikit-learn 0.19.1, la última versión estable en el momento de escribir este libro.

Uso de scikit-learn

El uso de scikit-learn se realiza a través de las distintas clases incluidas en la biblioteca Python `sklearn`. Esta biblioteca suele incluirse por defecto en las instalaciones de Anaconda, aunque siempre la podremos instalar usando el comando `pip` como se explica en el apéndice XX. Dentro de la biblioteca `sklearn` las distintas clases están divididas a su vez en paquetes diferenciados como `sklearn.neighbors` para algoritmos basados en cercanía de vecinos, `sklearn.linear_model` para modelos lineales o `sklearn.preprocessing` para las técnicas de preprocesado.

Como hemos comentado, el uso de scikit-learn es sencillo porque todas las tareas siguen el mismo patrón:

1. Crear un objeto de una clase concreta estableciendo sus parámetros. Este objeto puede ser un clasificador, un objeto que servirá para realizar regresión, un objeto que aplicará una etapa de preproceso a nuestros datos, etc.
2. Adecuar el objeto recién creado a los datos de entrenamiento. Este proceso se realiza siempre a través del método `fit`. En el caso de clasificación y regresión, el método `fit` recibirá dos parámetros: el conjunto `X` de *n* instancias (sin su clase) y por otro lado una secuencia `y` con los *n* valores de la clase. En el caso de análisis de grupos o preprocesado el método `fit` recibirá únicamente el conjunto `X` de instancias ya que no existe ningún valor de clase que utilizar. El método `fit` no devolverá nada, pero actualizará el estado interno del objeto usando los datos que ha observado.
3. Utilizar el objeto entrenado. Si estamos realizando clasificación o regresión, el objeto tendrá un método `predict` que recibe un conjunto `X` de instancias sin clase y devuelve una secuencia de clases predichas. Si estamos preprocesando datos, el objeto tendrá un método `transform` que recibe un conjunto de instancias y aplica el preprocesado configurado (por ejemplo, escalar un atributo o aplicar *one hot enconding*). En el caso de análisis de grupos, lo más usual es acceder a los atributos del objeto entrenado para obtener los centros de los clústeres encontrados, aunque también se puede invocar `transform` para asignar nuevas instancias a cada grupo.

En los objetos que realizan preprocesado de datos, lo usual es *entrenar* con el método `fit` y el conjunto de instancias `X` (por ejemplo, para que detecte el rango de valores de cada atributo) e inmediatamente después aplicar `transform` con el mismo conjunto `X` para que aplique las transformaciones adecuadas. En estos casos, scikit-learn proporciona un método `fit_transform` que entrena el objeto sobre un conjunto de instancias y posteriormente lo transforma, ahorrándonos un paso intermedio. Más adelante veremos que este método juega un papel importante en las *tuberías* de scikit-learn.

Preprocesado

Veamos con un ejemplo cómo aplicar distintas fases de preprocesado al conjunto de datos con los pasajeros del Titanic. Partiremos del `DataFrame` de `pandas` llamado `titanic` en el que habíamos eliminado algunas columnas y además habíamos codificado *Sex* y *Embarked* con números. Lo primero que vamos a hacer es dividir el conjunto completo en dos partes: el conjunto de entrenamiento (80%) y el conjunto de test (20%). Además, dividiremos cada uno de estos conjuntos a su vez para tener por un lado el valor de la clase y por otro el resto de atributos. Como esta transformación la vamos a realizar varias veces a lo largo del capítulo, vamos a definir

una función `split_label` que acepta un `DataFrame`, una proporción de test y el nombre de la clase y nos devuelve 4 conjuntos de datos:

```
from sklearn.model_selection import train_test_split

def split_label(df, test_size, label):
  train, test = train_test_split(df,test_size=test_size)
  features = df.columns.drop(label)
  train_X = train[features]
  train_y = train[label]
  test_X  = test[features]
  test_y  = test[label]
  return train_X, train_y, test_X, test_y
```

Utilizamos el método `train_test_split` para dividir el conjunto `df` en `train` y `test`, donde `test` tendrá una proporción `test_size` sobre el conjunto original. Cada uno de estos conjuntos será un `DataFrame` de `pandas`. Luego los separamos a su vez en dos fragmentos para quedarnos con la clase por un lado (`train_y` y `test_y`) y el resto de atributos por otro (`train_X` y `test_X`). Concretamente `train_X` y `test_X` serán objetos `DataFrame`, mientras que `train_y` y `test_y` serán objetos `Series`. La selección de atributos la realizamos filtrando las columnas del `DataFrame` directamente, tal y como vimos en este capítulo. Como `df.colums` nos devuelve un índice de Pandas con todas las columnas, únicamente debemos eliminar la columna almacenada en `label` invocando a `drop`.

Una vez tenemos nuestra función `split_label`, dividir `titanic` con una proporción de 0.2 (20%) para test y usando la columna *Survived* como clase es muy sencillo:

```
>>> train_X, train_y, test_X, test_y =
      split_label(titanic, 0.2, 'Survived')
```

Vamos a realizar dos pasos de preprocesado: realizar *one hot encoding* en la columna *Embarked* y posteriormente aplicar un escalado a todos los atributos para que sus valores estén en el rango [0,1]. Podríamos haber aplicado *one hot encoding* también a los atributos *Sex* y *Pclass*, pero no lo hemos hecho. En el caso de *Sex* porque se trata de un atributo categórico nominal con valores 0/1, así que la distancia entre esos dos valores es la adecuada y no tendría sentido dividirlo en dos atributos binarios a su vez. El caso de *Pclass* es diferente, ya que se trata de un atributo categórico ordinal donde la codificación 1-2-3 ya refleja adecuadamente la distancia entre distintas clases de billete. En casos así se puede dejar el atributo como está o aplicar *one hot encoding*, nosotros hemos preferido dejarlo tal cual

porque no hemos observado ninguna mejora significativa en el aprendizaje al aplicar *one hot enconding*.

Para aplicar *one hot encoding* utlizaremos la clase `OneHotEncoder` de la biblioteca Python `sklearn.preprocessing`. Si no se pasa ningún parámetro al construir el objeto codificador, este codificará todos los atributos. En nuestro caso queremos codificar únicamente el atributo *Embarked*, así que deberemos indicar el índice de dicho atributo a través del parámetro `categorical_features`. Adicionalmente, también indicaremos que queremos que el resultado sea un objeto de tipo `ndarray` en lugar de matrices usando el parámetro `sparse=False`:

```
>>> index_Embarked = train_X.columns.get_loc('Embarked')
>>> ohe = OneHotEncoder(
      categorical_features=[index_Embarked],
      sparse=False)
>>> train_X_1 = ohe.fit_transform(train_X)
```

Utilizamos el método `get_loc` para obtener el índice de la columna *Embarked* de nuestro `DataFrame`, que almacenamos en la variable `index_Embarked`. Esta opción es mejor que contar manualmente porque funcionará correctamente aunque reordenemos, añadamos o eliminemos columnas. Con este índice creamos un objeto `ohe` y posteriormente lo utilizamos para transformar `train_X`. Utilizamos el método `fit_transform` para que detecte los distintos valores de la columna y después transforme el conjunto de datos, generando el conjunto transformado `train_X_1`. El resultado es el mismo que aplicar primero `fit` y luego `transform` usando el mismo conjunto de datos. Es importante darse cuenta de que el resultado `train_X_1` de la transformación ya no es un `DataFrame` sino un `ndarray` de tamaño (569, 9) que contiene elementos de tipo `float64`.

Posteriormente realizamos el escalado de todos los atributos al rango [0,1]. Para ello utilizaremos la clase `MinMaxScaler` de la biblioteca Python `sklearn.preprocessing`. En este caso los parámetros por defecto son adecuados:

```
>>> min_max_scaler = MinMaxScaler()
>>> train_X_2 = min_max_scaler.fit_transform(train_X_1)
```

Como se puede observar, volvemos a aplicar `fit_transform` para convertir el conjunto `train_X_1` (la salida del codificador `ohe`) en el conjunto preprocesado `train_X_2`. En este momento ya tenemos las instancias de entrenamiento en su formato final para entrenar un clasificador. `train_X_2` es un objeto de tipo `ndarray` con tamaño (569, 9), es decir, 569 instancias de 9 atributos cada una. Originalmente, el `DataFrame titanic` tenía 8 columnas, pero hemos eliminado la columna *Survived*

y hemos codificado la columna *Embarked* en 3 columnas binarias. Además, cada atributo contiene un número real entre 0 y 1, tal y como podemos comprobar si mostramos las 3 primeras instancias de `train_X_2`:

```
>>> for i in range(3):
>>>   print(train_X_2[i])
[0.  0.  1.  0.5  1.  0.25860769  0.  0.  0.14346245]
[0.  1.  0.  1.   0.  0.27117366  0.  0.  0.01512699]
[0.  1.  0.  1.   0.  0.27117366  0.  0.  0.01512699]
```

Clasificación

Para realizar clasificación sobre el conjunto de pasajeros del Titanic lo primero que necesitamos es crear un objeto clasificador y entrenarlo. Scikit-learn dispone de una amplia colección de más de 20 algoritmos para clasificación, que se pueden consultar en la documentación de Scikit-learn (sección *Supervised learning*). En este caso concreto utilizaremos una técnica muy conocida llamada *máquinas de vectores de soporte* (*SVM* según sus siglas en inglés *support vector machines*). Esta técnica está diseñada para problemas de clasificación binaria, y trata de encontrar un hiperplano que separe los elementos de una y otra clase. Como en general pueden existir infinitos planos separadores, buscará aquel que esté más alejado de cada clase, maximizando el margen. En ocasiones es imposible encontrar un hiperplano de separación perfecta, así que la técnica SVM se puede extender para permitir que algunas instancias estén en el lado incorrecto a cambio de una penalización configurable. De la misma manera, también se pueden considerar *funciones kernel* que permiten transformar el espacio origen en un espacio de más dimensiones donde hay más posibilidades de encontrar el hiperplano separador. Para más detalles recomendamos consultar los libros de texto sobre aprendizaje automático incluidos en las referencias.

Para realizar clasificación usando SVM usaremos la clase `SVC` de scikit-learn. Para ello, crearemos un objeto con los parámetros por defecto e invocaremos su método `fit` con el conjunto de entrenamiento (`train_X_2` y `train_y`):

```
>>> from sklearn.svm import SVC
>>> clf = SVC()
>>> clf.fit(train_X_2, train_y)
```

A partir de este momento el objeto `clf` estará entrenado y nos servirá para clasificar nuevas instancias. Sin embargo, las instancias a clasificar deben tener los mismos 9 atributos que tenían las instancias de `train_X_2`. Esto no es ningún

problema porque disponemos de los transformadores ohe para *one hot encoding* y min_max_scaler para escalar al rango [0,1]. Si queremos transformar instancias directamente extraídas del DataFrame de pandas solo tendremos que pasarlas por estos transformadores en el mismo orden en el que fueron configurados:

```
>>> test_X_2 = min_max_scaler.transform(ohe.transform(test_X))
>>> clf.predict(test_X_2)
array([0, 1, 0, 1, 0, 0, 0, 1, (…),0, 0, 0, 0, 0, 0])
```

El conjunto de instancias test_X es transformado primero con ohe y luego con min_max_scaler, generando el conjunto de 9 atributos test_X_2. En ambos casos, los conjuntos tienen 143 instancias de test. Si invocamos al método predict nos devolverá un ndarray de 143 clases con las predicciones para cada instancia. Como disponemos de la secuencia test_y con los resultados reales podríamos calcular manualmente la tasa de aciertos o cualquier otra medida de calidad del modelo, aunque Scikit-learn ya dispone de muchas funciones para calcular métricas de calidad sobre modelos dentro del paquete sklearn.metrics. Sin embargo, los clasificadores de Scikit-learn también cuentan con un método score que recibe un conjunto de test y las clases esperadas y evalúa el modelo con la métrica estándar, en este caso la tasa de aciertos:

```
>>> clf.score(test_X_2, test_y)
0.8041958041958042
```

Como se puede ver, el clasificador SVC con los hiperparámetros por defecto ha conseguido una tasa de aciertos del 80% en nuestro conjunto de 143 instancias de test. No es un resultado excelente, pero sí bastante rentable para el esfuerzo que nos ha llevado. Si necesitásemos mejorarlo deberíamos considerar añadir nuevos atributos a nuestro conjunto, realizar una búsqueda de los mejores hiperparámetros con un conjunto de validación o probar con otro clasificador. Como todos los clasificadores tienen la misma interfaz (fit, predict, score) cambiar de clasificador solo nos requerirá cambiar la línea en la que creamos el objeto clasificador. Por ejemplo, podemos usar clasificación siguiendo la técnica de los *k* vecinos más cercanos de la siguiente manera:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> clf = KNeighborsClassifier()
>>> clf.fit(train_X_2, train_y)
>>> clf.score(test_X_2, test_y)
0.7762237762237763
```

Regresión

Aplicar regresión sobre los datos de los pasajeros del Titanic usando scikit-learn es muy similar a aplicar clasificación, solo que seleccionando un atributo continuo como clase. En este caso vamos a tratar de predecir la tarifa pagada (*Fare*) a partir del resto de atributos. Esto modificará la manera de separar atributos y clase a partir de nuestro conjunto original `titanic` para seleccionar la columna *Fare* tanto en `train_y` como en `test_y`. Esto es muy sencillo gracias a la función `split_label` definida anteriormente:

```
>>> train_X, train_y, test_X, test_y =
      split_label(titanic, 0.2, 'Fare')
```

Una vez tenemos los conjuntos de entrenamiento y test separados en atributos y clase, tendremos que realizar las mismas transformaciones que en el caso de la clasificación para aplicar *one hot encoding* y escalar los valores numéricos al rango [0,1]. Obsérvese que como estamos eligiendo otra columna como clase, el valor de `index_Embarked` será distinto del caso de clasificación, por lo que tendremos que volver a calcularlo. Aprovechamos también para transformar el conjunto de test `test_X` y obtener `test_X_2` usando los mismos transformadores:

```
>>> index_Embarked = train_X.columns.get_loc('Embarked')
>>> ohe = OneHotEncoder(
            categorical_features=[index_Embarked],
            sparse=False)
>>> train_X_1 = ohe.fit_transform(train_X)
>>> min_max_scaler = MinMaxScaler()
>>> train_X_2 = min_max_scaler.fit_transform(train_X_1)
>>> test_X_2  = min_max_scaler.transform(
                 ohe.transform(test_X))
```

Scikit-learn dispone de muchos algoritmos para realizar regresión sobre conjuntos de datos, que se pueden consultar en la sección *Supervised learning* de su documentación. En este caso vamos a ver cómo utilizar una de las más simples: *regresión lineal*. Este tipo de regresión toma la suposición de que la clase *y* de una instancia con *n* atributos $(x_1, x_2, \ldots, x_n)$ se calcula con la siguiente ecuación lineal:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

La tarea de la regresión lineal es encontrar aquellos valores $(w_0, w_1, w_2, \ldots, w_n)$ que minimicen la diferencia entre la clase calculada y la clase real sobre el conjunto de entrenamiento. Se trata por tanto de un problema de optimización (minimización

en este caso) cuya solución $(w_0, w_1, w_2 \ldots, w_n)$ nos permitirá clasificar futuras instancias simplemente usando la ecuación anterior.

Para realizar regresión lineal en scikit-learn utilizaremos la clase `LinearRegression` de la biblioteca `sklearn.linear_model`. Esta clase admite varios parámetros en su constructora, pero como en los ejemplos anteriores vamos a utilizar los valores por defecto. Una vez tenemos el objeto para realizar regresión solo tenemos que entrenarlo con el método `fit` y comprobar la calidad del modelo generado usando el conjunto de test mediante el método `score`:

```
>>> from sklearn.linear_model import LinearRegression
>>> reg = LinearRegression()
>>> reg.fit(train_X_2, train_y)
>>> reg.score(test_X_2, test_y))
0.2743815749934385
```

En este caso el método `score` de `LinearRegression` calcula el coeficiente de determinación R^2. El mejor valor de esta métrica es 1, pero puede tomar valores negativos para modelos con escasa calidad. En este caso el valor de 0,27 no parece muy bueno, pero tampoco excesivamente malo. Si esta métrica por defecto no nos gusta, siempre podemos utilizar cualquier otra de las disponibles en la biblioteca `sklearn.metrics` como el error cuadrático medio (invocando a la función `mean_squared_error`) o el error absoluto medio (invocando a la función `mean_absolute_error`). Para llamar a estas funciones necesitamos pasar como parámetros los valores reales (`test_y`) y los valores predichos (`pred`) para que calculen las diferencias:

```
>>> from sklearn.metrics import mean_squared_error,
      mean_absolute_error
>>> pred = reg.predict(test_X_2)
>>> mean_squared_error(test_y, pred)
756.4492780821876
>>> mean_absolute_error(test_y, pred)
20.144025174825174
```

Como en el caso de la clasificación, el método `predict` generará un `ndarray` de 143 valores, en este caso números reales. Según el error absoluto medio la regresión lineal obtiene resultados que se alejan unas 20 libras del precio real. Considerando que los precios de los billetes toman valores entre 0 y 513 libras, parece un modelo suficientemente preciso. Sin embargo, una diferencia de 20 libras en 1912 puede ser demasiado elevada. Al igual que con clasificación, podríamos hacer pruebas con distintas técnicas de regresión para buscar mejores modelos simplemente cambiando la construcción del objeto `reg`, por ejemplo usando regresión mediante *k*

vecinos más cercanos (clase `KNeighborsRegressor` de la biblioteca `sklearn.neighbors`). El resto del código funcionaría exactamente igual gracias a la uniformidad en la interfaz de las clases.

Análisis de grupos

El último ejemplo de aprendizaje automático que vamos a realizar sobre el conjunto de datos de los pasajeros del Titanic será dividirlos en conjuntos de pasajeros similares, es decir, realizar análisis de grupos. Scikit-learn dispone de varios algoritmos para realizar análisis de grupos, que se pueden encontrar en la sección *Unsupervised learning* de su documentación. En este caso vamos a considerar uno de los algoritmos más utilizados para realizar análisis de grupos: *k-means*.

k-means es un algoritmo iterativo que particiona las instancias en *k* grupos disjuntos, donde el valor *k* debe ser fijado por el usuario. Este algoritmo realiza una división inicial en *k* grupos, y la va refinando iterativamente hasta que los grupos obtenidos no cambian. En cada iteración calcula el *centroide* del grupo, es decir, la instancia promedio que en cada atributo toma como valor la media aritmética de los valores de dicho atributo en todas las instancias del grupo. Una vez ha calculado los *k* centroides, que puede verse como un punto en un espacio n-dimensional, divide las instancias en los *k* grupos asignando cada una al centroide más cercano. A pesar de su simplicidad, es un algoritmo que produce buenos resultados, aunque es necesario encontrar un valor de *k* adecuado para cada conjunto.

Aplicar *k-means* a los pasajeros del Titanic es aún más sencillo que aplicar clasificación o regresión, ya que no tenemos un atributo clase que debamos separar. Y como no tenemos clase, no es necesario dividir el conjunto entre entrenamiento y test. Lo que sí que necesitaremos es aplicar *one hot encoding* y escalar los atributos al rango [0,1]. Esto se realiza usando las clases de scikit-learn que ya conocemos, aunque ahora directamente a partir del `DataFrame titanic`.

```
>>> index_Embarked = titanic.columns.get_loc('Embarked')
>>> ohe = OneHotEncoder(
            categorical_features=[index_Embarked],
            sparse=False)
>>> titanic_1 = ohe.fit_transform(titanic)
>>> min_max_scaler = MinMaxScaler()
>>> titanic_2 = min_max_scaler.fit_transform(titanic_1)
```

Una vez tenemos los datos del Titanic preprocesados, realizar el análisis de grupos es similar a aplicar cualquier otro algoritmo de aprendizaje automático, solo

debemos crear un objeto KMeans con los parámetros adecuados y entrenarlo con fit:

```
>>> from sklearn.cluster import KMeans
>>> clu = KMeans(n_clusters=3)
>>> clu.fit(titanic_2)
>>> clu.cluster_centers_
[ [ 1.22124533e-15  5.39083558e-02 9.46091644e-01
    -2.60902411e-15 7.77628032e-01 8.49056604e-01
    3.75477998e-01  1.11051213e-01 6.01976640e-02
    3.85185852e-02 ]
  (…)
]
```

La clase KMeans está en la biblioteca sklearn.cluster, y su constructora admite varios parámetros. En este caso únicamente hemos utilizado n_cluster para que divida el conjunto en 3 grupos. El resultado del entrenamiento serán los centroides de cada uno de los 3 grupos, que podremos obtener a través del atributo cluster_centers_. Este atributo almacena una lista de 3 centroides, cada uno un punto de 10 dimensiones. El número de dimensiones es el esperado, ya que el conjunto original titanic tenía 8 columnas, y una de ellas (*Embarked*) la hemos multiplicado por 3 al aplicar *one hot encoding*. A partir de los centroides podríamos recorrer el conjunto titanic_2 y detectar manualmente a qué clúster pertenece cada pasajero. Sin embargo, esto no es necesario, ya que el objeto clasificador almacena esa información en el atributo labels_. Como el conjunto de datos titanic_2 tenía 712 instancias, labels_ contendrá un ndarray de NumPy con 712 números tomando valores 0, 1 o 2 dependiendo del clúster al que pertenezca:

```
>>> clu.labels_
array([0, 2, 1, 1, 0, 0, 0, 1, 2, 1,(…), 2, 0], dtype=int32)
```

Usando este atributo del clasificador podremos calcular métricas que nos permitan conocer la calidad del agrupamiento generado. Para ello utilizaremos el coeficiente de silueta y el índice Calinski-Harabasz, ambos en la biblioteca sklearn.metric:

```
>>> from sklearn.metrics import silhouette_score,
      calinski_harabaz_score
>>> silhouette_score(titanic_2, clu.labels_)
0.39752927745607614
>>> calinski_harabaz_score(titanic_2, clu.labels_)
360.02042405738507
```

En el caso de métricas de calidad de agrupamientos, no siempre es sencillo interpretar los valores. El coeficiente de silueta toma valores entre -1 (peor valor) y 1 (mejor valor), por lo que un coeficiente de 0,4 puede parecer adecuado. Sin embargo, el índice Calinski-Harabasz mide la proporción entre la dispersión intra-clúster y la dispersión inter-clúster, así que salvo para comparar distintos agrupamientos el valor obtenido no nos proporciona mucha información por sí solo.

Por último, indicar que el modelo obtenido nos permite procesar nuevas instancias y asignarlas a uno de los 3 grupos. Para ello, tendríamos que invocar al método `transform` pasando como parámetro un conjunto de instancias con 10 atributos cada una. El resultado sería un `ndarray` con tantas filas como instancias y por cada una de ellas 3 valores, la distancia al centroide de cada grupo.

Otros aspectos de scikit-learn

Para finalizar el apartado de scikit-learn queremos introducir brevemente tres aspectos de la biblioteca que son muy útiles: la creación de tuberías, la persistencia de los modelos y la optimización de hiperparámetros.

TUBERÍAS

En los ejemplos que hemos visto, a la hora de realizar aprendizaje supervisado teníamos que dividir el conjunto de datos en uno de entrenamiento y otro de test. Luego aplicábamos algunos objetos transformadores al conjunto de entrenamiento para codificar y escalar atributos en el conjunto de entrenamiento y realizábamos el entrenamiento. Sin embargo, antes de poder usar el modelo sobre los datos de test debíamos recordar aplicar las mismas transformaciones en el mismo orden, o si no la invocación a `predict` fallaría. En este caso era sencillo porque teníamos únicamente dos transformaciones, pero podríamos haber tenido más y habríamos tenido que recordar el orden entre ellas.

Para simplificar este proceso, scikit-learn proporciona tuberías (*pipelines* en inglés). Una tubería es una secuencia de objetos transformadores que finaliza (opcionalmente) en un objeto que realiza aprendizaje automático. Si el último elemento es también un transformador se tratará de una *tubería de transformación*, en otro caso se tratará de una *tubería de aprendizaje automático*. En esta secuencia, todos los objetos salvo el último deben tener un método `fit_transform`. A la hora de entrenar una tubería de entrenamiento con el método `fit`, se invocará en secuencia a los métodos `fit_transform` de todos los objetos salvo el último. La salida de este último transformador será la que se use para invocar al método `fit` del objeto final de la tubería.

Veamos cómo construir una tubería para clasificar los pasajeros del Titanic usando SVM. Como hemos visto en este capítulo, necesitamos 3 etapas:

1. *One hot encoding* para codificar la columna *Embarked*.
2. Escalado de todos los atributos al rango [0,1].
3. Entrenamiento de un objeto `SVC`.

Lo primero que necesitaremos son los conjuntos de entrenamiento y test convenientemente divididos, para ellos usamos nuestra función `split_label` para seleccionar el 20% de las instancias para test y elegir como clase la columna *Survived*:

```
>>> train_X, train_y, test_X, test_y =
      split_label(titanic, 0.2, 'Survived')
```

Para construir la tubería necesitamos crear los 3 objetos con sus correspondientes parámetros. Este proceso es similar al que hemos visto anteriormente:

```
>>> index_Embarked = train_X.columns.get_loc('Embarked')
>>> ohe = OneHotEncoder(
      categorical_features=[index_Embarked],
      sparse=False)
>>> min_max_scaler = MinMaxScaler()
>>> svm = SVC()
```

El siguiente paso es crear un objeto de tipo `Pipeline` de la biblioteca `sklearn.pipeline` y establecer el orden de los objetos y su nombre:

```
>>> pipe = Pipeline([('ohe', ohe),
                     ('sca', min_max_scaler),
                     ('clf', svm)])
```

Para construir la tubería pasamos una lista de parejas *(nombre, objeto)* indicando el orden exacto de las etapas. El nombre de las etapas nos permitirá más adelante acceder a una etapa concreta mediante su nombre, por ejemplo `pipe.named_steps['clf']` nos devolvería el objeto de la clase `SVC`. Una vez tenemos la tubería creada, la podemos utilizar directamente como si de un objeto clasificador se tratara:

```
>>> pipe.fit(train_X, train_y)
>>> pipe.score(test_X, test_y)
0.7762237762237763
>>> pipe.predict(test_X)
array([0, 0, 0, 0, 0, 1, (…), 0,  0, 1, 0, 0, 0, 1, 0])
```

Obsérvese cómo a la hora de obtener la tasa de aciertos con `score` o de predecir con `predict` no hemos necesitado transformar el conjunto `test_X`. Como la tubería está configurada y entrenada, automáticamente encadena invocaciones `transform` en todos los objetos salvo el último, en el que invocará `score` o `predict` sobre el resultado de la última etapa. En general es recomendable utilizar tuberías a la hora de usar scikit-learn ya que simplifican el uso y evitan errores. La figura 5-4 muestra la cadena de invocaciones sobre cada una de las etapas de la tubería que se genera al invocar los métodos `pipe.fit` y `pipe.score` del ejemplo anterior.

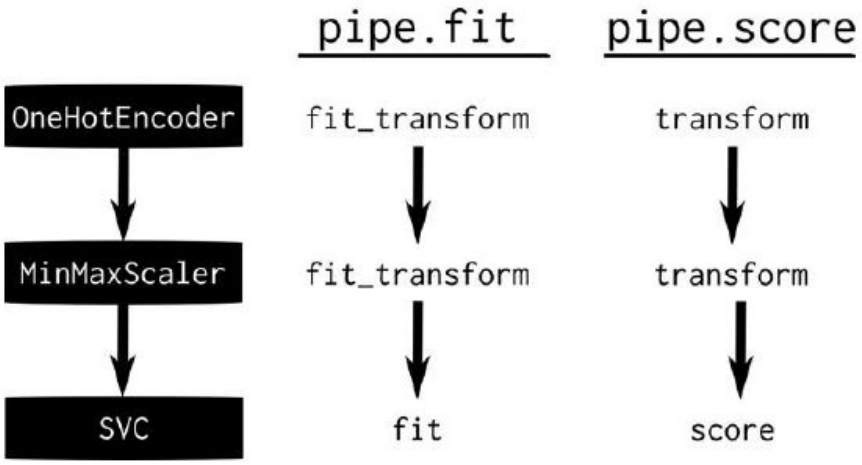


Figura 5-4. Métodos invocados en una tubería.

PERSISTENCIA DE MODELOS

En los ejemplos que hemos visto entrenábamos un objeto y lo utilizábamos inmediatamente después para predecir la clase en un conjunto de test o para asignar un grupo a nuevas instancias. Como el conjunto de datos sobre los pasajeros del Titanic es pequeño, repetir el entrenamiento cada vez es algo factible. Sin embargo, normalmente trabajaremos con conjuntos de datos más grandes donde el tiempo dedicado al entrenamiento puede necesitar minutos o incluso varias horas. En esos casos querremos almacenar el modelo entrenado y recuperarlo en el futuro, cuando dispongamos de instancias nuevas para predecir o asignar grupo.

Python dispone de una biblioteca estándar llamada `pickle` para serializar objetos, es decir, para convertirlos en una secuencia de bytes que se pueden almacenar en disco o enviar por la red. Los modelos de scikit-learn se pueden volcar a ficheros y recuperarlos usando `pickle`, sin embargo, recomiendan el uso de la biblioteca `joblib` del paquete `sklearn.externals`. Esta biblioteca es más eficiente al tratar con objetos que almacenan gran cantidad de estructuras de NumPy como es el caso de las clases de scikit-learn.

Veamos cómo se podría volcar a un fichero el modelo de *k-means* que entrenamos en la sección sobre análisis de grupos usando el método `dump`:

```
>>> clu = KMeans(n_clusters=3)
>>> clu.fit(titanic_2)
>>> joblib.dump(clu, 'data/kmeans.pkl')
```

Este método acepta un objeto y una ruta, y vuelca el objeto a un fichero, en este caso `data/kmeans.pkl`. El fichero generado para este modelo es bastante pequeño, de apenas 4 KB. Recuperarlo desde el fichero es igual de sencillo:

```
>>> loaded_clu = joblib.load('data/kmeans.pkl')
```

A partir de este momento podremos usar `loaded_clu` exactamente igual que el modelo recién entrenado, por ejemplo, para medir su calidad con el coeficiente de silueta o calcular la distancia a cada centroide para un conjunto de datos nuevo usando `transform`.

El volcado y la recuperación de objetos no se limitan a modelos entrenados, sino que se pueden aplicar a tuberías completas. De esta manera no tendremos que salvar cada etapa por separado y cargarlas, sino que todo ese proceso se realizará de manera automática. Por ejemplo, salvar la tubería que realiza análisis de grupos sería tan sencillo como:

```
>>> index_Embarked = titanic.columns.get_loc('Embarked')
>>> ohe = OneHotEncoder(
            categorical_features=[index_Embarked],
            sparse=False)
>>> sca = MinMaxScaler()
>>> clu = KMeans(n_clusters=3)
>>> pipe = Pipeline([('ohe', ohe),
                     ('sca', sca),
                     ('clu',clu)])
>>> pipe.fit(titanic)
>>> joblib.dump(pipe, 'data/kmeans_pipeline.pkl')
```

En este caso el fichero generado es ligeramente más pesado, de 6 KB, pero contiene todas las etapas de transformación y clasificación ya entranadas. Cargar la tubería únicamente requiere invocar a la función `load`:

```
>>> loaded_pipe = joblib.load('data/kmeans_pipeline.pkl')
```

OPTIMIZACIÓN DE HIPERPARÁMETROS

En todos los ejemplos de este capítulo hemos creado los objetos de scikit-learn como `SVC`, `LinearRegression` y `KMeans` con los parámetros por defecto. Esto no es muy buena idea, ya que cada uno de ellos tiene una cantidad elevada de parámetros a configurar que pueden tener un impacto importante en la calidad del modelo generado. ¿Cómo se pueden encontrar los mejores parámetros? Para ello se deben entrenar distintos con diferentes parámetros y quedarse con el que mejor calidad tenga frente el conjunto de validación que mencionamos al presentar el aprendizaje automático.

Para facilitar esta tarea, scikit-learn proporciona la clase `GridSearchCV` dentro del paquete `sklearn.model_selection`. Esta clase toma un objeto, lo entrena para distintas combinaciones de parámetros y se queda con el mejor modelo. Para ello es necesario crear un diccionario con los valores que queremos probar para cada parámetro. Para medir la calidad de los modelos utiliza internamente validación cruzada de 3 iteraciones, aunque se puede configurar el número de iteraciones o incluso utilizar una validación personalizada.

Si quisiéramos optimizar los hiperparámetros del clasificador `SVC` para nuestro conjunto de pasajeros del Titanic tendríamos que ejecutar el siguiente código (omitimos la codificación y el escalado):

```
>>> svc = svm.SVC()
>>> parameters = {'kernel': ['linear', 'rbf'],
                  'C':[1,2] }
>>> clf = GridSearchCV(svc, parameters)
>>> clf.fit(train_X, train_y)
```

Hemos creado un diccionario indicando que queremos probar dos valores del hiperparámetro `kernel` y otros dos del hiperparámetro `C`. Al invocar a `fit`, se probarán todas las combinaciones posibles con los valores del diccionario y se elegirá el mejor. Este proceso puede tardar bastante tiempo debido a la explosión combinatoria, en este caso probando 4 combinaciones y para cada una de ellas 3 entrenamientos debido a la validación cruzada de 3 iteraciones. Así que en general hay que tener cuidado al utilizar la clase `GridSearchCV` probando únicamente los hiperparámetros y valores más prometedores.

El objeto `clf` generado nos permite conocer cuáles han sido los hiperparámetros que generan el mejor modelo a través del atributo `best_params_`, o acceder directamente a ese mejor modelo a través del atributo `best_estimator_`. Sin

embargo, también se comporta como un objeto clasificador usando para ello el mejor modelo encontrado. El siguiente código muestra un ejemplo de su uso:

```
>>> print(clf.best_params_)
{'C': 1, 'kernel': 'linear'}
>>> print(clf.best_estimator_)
SVC(C=1, cache_size=200, class_weight=None,
coef0=0.0, decision_function_shape='ovr', degree=3,
gamma='auto', kernel='linear', max_iter=-1,
probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)
>>> clf.predict(test_X)
array([1, 0, 0, 0, 0, 0, 1, 1, 0,(…),  0, 1, 0, 0, 0, 1])
```

CONCLUSIONES

En este capítulo hemos visto cómo la clase `DataFrame` de `pandas` nos permite inspeccionar y transformar conjuntos de datos tabulares de manera muy sencilla. También hemos tratado con detalle el uso de scikit-learn para realizar aprendizaje automático sobre conjuntos de datos, lo que nos permite crear modelos y extraer conocimiento de ellos. La biblioteca scikit-learn dispone de un gran catálogo de algoritmos de aprendizaje automático y además es sencilla de utilizar, por lo que parece la solución ideal para cualquier tipo de problema. Sin embargo, tiene un problema: no está diseñada para escalar. Esto implica que todo el proceso de aprendizaje está restringido a una máquina, lo que limita fuertemente la cantidad de datos que se podrán manejar. Existen técnicas para mitigar esta limitación como realizar aprendizaje incremental sobre fragmentos del conjunto de datos más manejables, o lanzar distintos procesos que se ejecuten en los distintos núcleos de la CPU (algunas clases de scikit-learn admiten un parámetro `n_jobs` para esta tarea). Pero al final, si queremos manejar problemas *big data* reales con cientos de gigabytes o terabytes, scikit-learn no es la solución.

Dada la popularidad de las bibliotecas NumPy, Pandas y scikit-learn, ha aparecido una biblioteca que sigue sus mismas ideas y las adapta a entornos distribuidos. Esta biblioteca Python, llamada *Dask*, proporciona un mecanismo de cómputo distribuido que puede escalar a clústeres con cientos o miles de nodos. Además, proporciona versiones distribuidas de los arrays multidimensionales de NumPy y de los `DataFrame` de `pandas`, además de adaptar algoritmos de aprendizaje automático de scikit-learn (conocido como *Dask-ML*). Sin embargo, Dask no es la única alternativa para manejar problemas *big data* en Python. Existen varios sistemas de cómputo distribuido que soportan Python, y entre ellos podemos destacar Apache Spark por

su robustez, facilidad de uso e interconexión con el ecosistema de cómputo distribuido Hadoop. Al realizar soluciones *big data* no es común estar circunscrito a un lenguaje de programación, sino que es más provechoso tener la oportunidad de cooperar con todo un ecosistema de herramientas y decidir en cada momento cuál es la más adecuada para cada tarea. Además, Spark es un sistema con una trayectoria más larga que Dask, y que es utilizado por muchas empresas punteras en *big data*. Por todo ello, en los siguientes dos capítulos veremos cómo utilizar Spark desde Python para procesar conjuntos de datos y realizar aprendizaje automático en entornos 100% distribuidos.

REFERENCIAS

- Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (Second Edition). Wes McKinney. O'Reilly, 2017.
- Python Data Analytics Data Analysis and Science Using Pandas, matplotlib, and the Python Programming Language. Fabio Nelli. Appress, 2015.
- Documentación de Pandas: `https://pandas.pydata.org/pandas-docs/stable/`
- Hands-On Machine Learning with Scikit-Learn and TensorFlow: Con cepts, Tools, and Techniques to Build Intelligent Systems. Aurélien Géron. O'Reilly Media, 2017.
- Learning Scikit-Learn: Machine Learning in Python. Raul Garreta, Guillermo Moncecchi. Packt Publishing, 2013.
- Documentación de Scikit-learn: `http://scikit-learn.org/stable/documentation.html`
- Dask: Scalable analytics in Python `https://dask.pydata.org`

Libros sobre aprendizaje automático:

- Principles of Data Mining (3rd Edition). Max Bramer. Springer, 2016.
- Pattern Recognition and Machine Learning. Christopher M. Bishop. Springer, 2006.
- Machine Learning. Tom M. Mitchell. McGraw-Hill, 1997.
- The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd Edition). Trevor Hastie, Robert Tibshirani y Jerome Friedman. Springer, 2009.

6 PROCESAMIENTO DISTRIBUIDO CON SPARK

INTRODUCCIÓN

En el capítulo anterior vimos cómo utilizar la estructura `DataFrame` de la biblioteca `pandas` para almacenar información en forma de tablas, y cómo procesar estos datos para extraer conocimiento usando algoritmos de aprendizaje automático de la biblioteca `scikit-learn`. Sin embargo, estas bibliotecas no han sido diseñadas para ejecutarse en entornos *Big Data* donde la información se distribuye entre varios ordenadores interconectados, lo que impone una limitación a la cantidad de datos que pueden almacenar y procesar a un único ordenador o a un único núcleo del procesador. En este capítulo presentaremos *Apache Spark*, un sistema que nos permitirá almacenar y procesar grandes cantidades de datos de manera totalmente distribuida.

Apache Spark es un sistema de cómputo masivo diseñado para procesar datos de manera distribuida sobre clústeres de ordenadores. Gracias a su diseño distribuido, Spark puede procesar cantidades de datos del orden de terabytes o incluso petabytes (1000 terabytes).

A diferencia de otros mecanismos distribuidos de cómputo como *MapReduce*, que fue diseñado por Google para calcular la relevancia de los sitios web en su buscador, Spark trata de minimizar el acceso a disco. Podemos decir que en MapReduce se concibe el clúster como un disco gigante, la unión de los discos de todos los ordenadores del clúster, mientras que en Spark nos imaginamos el clúster como una

memoria gigante, la memoria resultante de combinar las memorias de todos los ordenadores del clúster. Al priorizar el uso de la memoria, que es bastante más rápida que los accesos a disco, consigue un rendimiento hasta unas 100 veces mayor que MapReduce para algunas tareas como la clasificación mediante regresión logística. Por otro lado, Spark surge con la idea de ser un sistema más flexible que MapReduce, permitiendo a sus usuarios realizar cómodamente cómputos que requieran un número arbitrario de transformaciones sobre sus datos.

Apache Spark es un sistema que surgió en el ámbito académico en el año 2009, concretamente en el laboratorio AMPLab de la Universidad de California en Berkeley, Estados Unidos. Desde ese momento, y gracias a su licencia de código abierto, empezó a ganar popularidad, y ya en 2014 se convirtió en un proyecto de primer nivel de la *Apache Software Foundation*. Actualmente, Apache Spark es uno de los sistemas de cómputo masivo más populares, conviviendo con el maduro mecanismo de cómputo MapReduce, así como con otros sistemas más nuevos como Apache Flink, Apache Storm o Apache Beam. La versión actual en el momento de escribir este libro es la 2.3.0, publicada el 20 de febrero de 2018. En el apéndice se pueden encontrar las instrucciones para instalarlo en modo local tanto en sistemas Windows como Linux y Mac.

Apache Spark está implementado en el lenguaje de programación Scala, que combina los paradigmas imperativo y funcional y es ejecutado en la máquina virtual de Java. Sin embargo, además de Scala, Spark proporciona interfaces de programación de aplicaciones (APIs en inglés) para Java, Python y R. Al ser ejecutado en la máquina virtual de Java, las APIs para Scala y Java suelen producir mejores rendimientos. De la misma manera, las nuevas características y mejoras se suelen introducir primero en estas dos APIs y posteriormente se van portando a Python y R. En este aspecto, es en el apartado de algoritmos de aprendizaje máquina donde ese retraso se nota más.

El sistema Spark está formado por distintos componentes que descansan sobre un núcleo que proporciona la funcionalidad básica. El esquema general tiene el aspecto que se muestra en la figura 6-1.

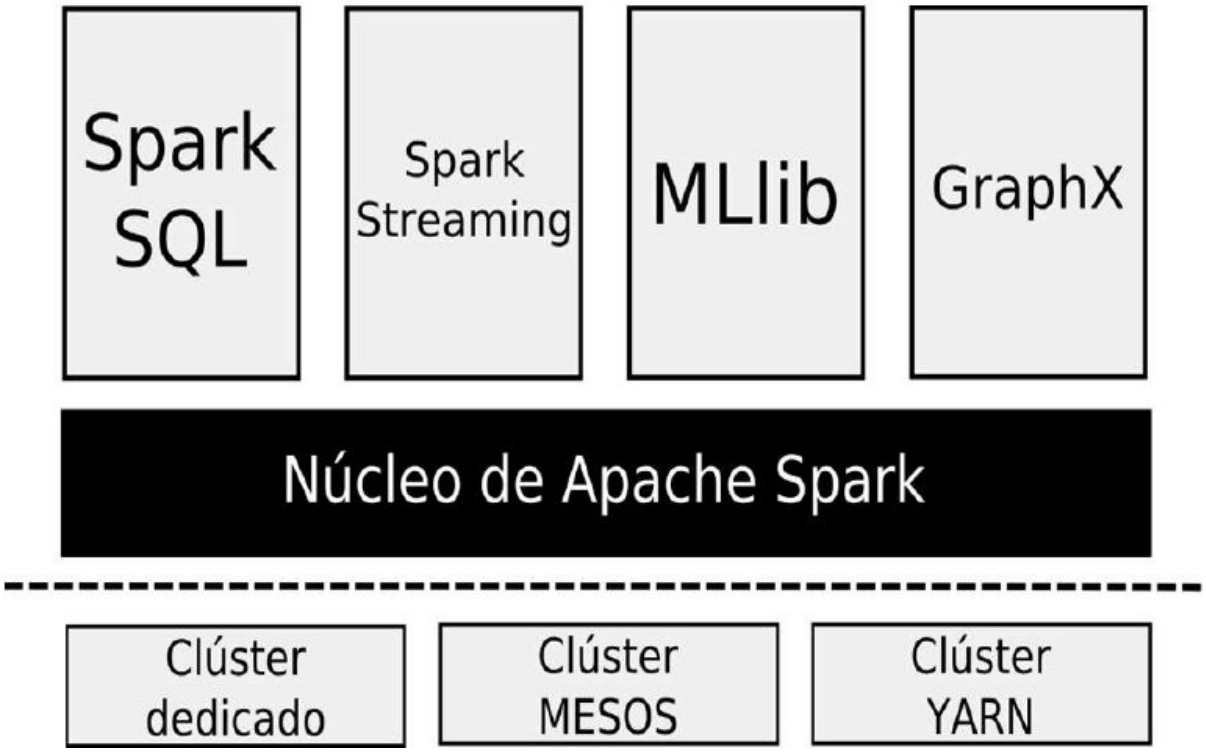


Figura 6-1. Estructura de Apache Spark.

El núcleo de Spark permite tratar los llamados *conjuntos de datos distribuidos resilientes*, el tipo de datos básico de Spark. Los datos de estos conjuntos se encuentran distribuidos a lo largo del clúster, por lo que este componente hace uso del gestor de recursos del clúster. En el caso de clústeres dedicados únicamente a ejecutar programas Spark, se puede utilizar el gestor autónomo (*standalone*), pero también se puede hacer uso de los gestores Mesos y YARN para ejecutar programas Spark sobre clústeres que soportan otros servicios y sistemas. Esta última opción es la más flexible, ya que permite compartir un clúster entre distintos sistemas de cómputo y elegir en cada momento el que mejor se adecue a las necesidades.

Sobre el núcleo de Spark podemos distinguir varios componentes:

- *Spark SQL* es el componente que proporciona los `DataFrames`, estructuras de datos tabuladas similares a tablas en bases de datos relacionales, que simplifican el manejo de datos masivos. Estas estructuras se pueden operar directamente mediante instrucciones con sintaxis SQL, y también proporcionan métodos y funciones alternativos con similar funcionalidad.
- *Spark Streaming* es el componente Spark para tratar flujos de datos constantes que se deben procesar en tiempo real. Para ello, considera estos flujos como una sucesión de pequeños lotes que se van procesando en cada ventana temporal.

- *MLlib* es el componente que alberga todo lo relacionado con el aprendizaje automático. MLlib proporciona dos APIs a los programadores: la original que está basada en los conjuntos de datos distribuidos resilientes, y una API evolucionada (conocida comúnmente como SparkML aunque sea parte de MLlib) que está basada en los `DataFrames`. Desde la versión 2.0 de Spark el API original ha entrado en fase de mantenimiento, por lo que no recibirá nuevas características sino únicamente correcciones de errores. Además, se espera que esta API desaparezca completamente a partir de la versión 3.0. Dado que SparkML proporciona una interfaz más amigable y de más alto nivel que el API original, se recomienda su utilización, y será el API que presentemos en este libro.
- *GraphX* es el componente Spark para el manejo de grafos que pueden almacenar propiedades tanto en los nodos como en las aristas. GraphX proporciona algoritmos de grafos como *PageRank* o la detección de componentes conexas. Es un componente que ha recibido menos desarrollo que los demás, y por tanto no cuenta con demasiadas funcionalidades.

A la hora de ejecutar un programa Spark de manera distribuida, podemos distinguir dos tipos de procesos: el proceso *driver* y los procesos *executor*. El proceso *driver* es el proceso principal que lanza el programa. Este proceso tiene un objeto `SparkContext` que le permite conectar con el gestor del clúster y reservar procesos *executor* en los distintos nodos del clúster. Cada uno de los nodos del clúster (también conocidos como nodos *worker*) podrá ejecutar uno o varios procesos *executor*, que almacenarán fragmentos de los datos del programa y realizarán operaciones sobre ellos. Normalmente, un nodo del clúster ejecutará tantos procesos *executor* como núcleos tenga su procesador. Durante la ejecución del programa el proceso *driver* irá enviando peticiones a los distintos procesos *executor*, que pueden contactar entre ellos para realizar algunas tareas y comunicar con el proceso *driver* para devolver resultados. En la figura 6-2 se puede ver un ejemplo de un proceso *driver* conectado a 3 procesos *executor* localizados en dos nodos del clúster.

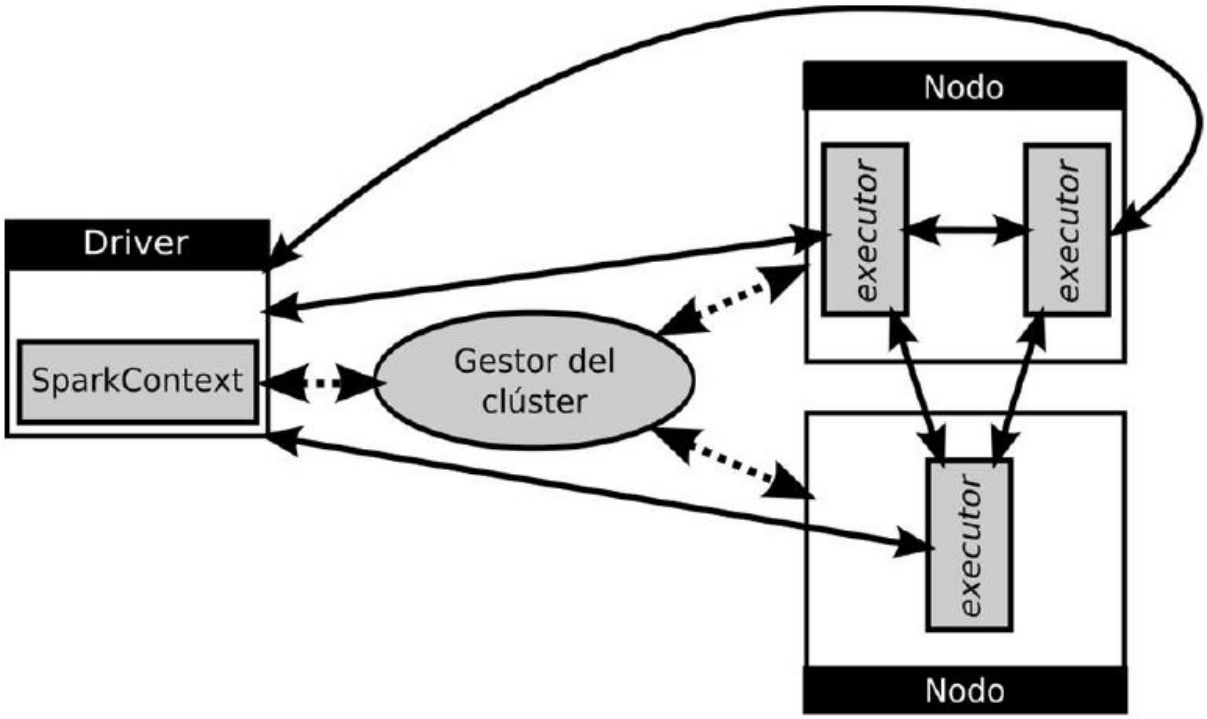


Figura 6-2. Proceso driver conectado a tres procesos executor.

El objeto de este capítulo son los conjuntos de datos distribuidos resilientes y sus operaciones, es decir, el corazón de Spark sobre el que está construido el resto de componentes. Nos centraremos en cómo se pueden crear estos conjuntos de datos y en los dos tipos de operaciones que permiten: acciones y transformaciones. Para finalizar, mostraremos un ejemplo compuesto por varias operaciones para preprocesar el conjunto de datos de los pasajeros del Titanic que vimos en el capítulo anterior. En el siguiente capítulo presentaremos el tipo de datos `DataFrame` del componente SparkSQL, además de su análisis mediante algoritmos de aprendizaje automático del componente SparkML.

CONJUNTOS DE DATOS DISTRIBUIDOS RESILIENTES

Los conjuntos de datos distribuidos resilientes (en lo sucesivo RDDs por sus siglas en inglés: *Resilient Distributed Dataset*) constituyen el tipo de datos básico de Spark. Estos conjuntos almacenan la información de manera distribuida entre todos los equipos del clúster. Durante la ejecución de un programa Spark se construyen varios RDDs que se dividen en distintos fragmentos y son almacenados (prioritariamente) en la memoria de los equipos del clúster.

Podemos destacar 4 características de los RDDs. La primera característica es que están formados por un conjunto de registros, también llamados elementos, todos del

mismo tipo. Por ejemplo, si cargamos un RDD a partir de un fichero de texto se creará un RDD de cadenas de texto, una por cada línea del fichero, tal y como se puede ver en la figura 6-3.

“Muchos años después, frente al pelotón de fusilamiento, “
“el coronel Aureliano Buendía había de recordar aquella tarde remota “
“en que su padre lo llevó a conocer el hielo. “
“Macondo era entonces una aldea de 20 casas de barro y cañabrava “
“construidas a la orilla de un río de aguas diáfanas que se precipitaban ”
“por un lecho de piedras pulidas, blancas y enormes como huevos prehistóricos. “

Figura 6-3. RDD de cadenas de texto.

En lenguajes estáticamente tipados como Scala o Java, al declarar un RDD se debe definir el tipo de los registros que almacenará para que el compilador pueda detectar errores, por ejemplo, con las expresiones `RDD[String]` en Scala o `JavaRDD<String>` en Java. En cambio Python, al ser un lenguaje con tipado dinámico, nos permite crear RDDs heterogéneos, de la misma manera que nos permite crear listas que mezclan distintos tipos como `[1, True, "hola", 3.14]`. A pesar de esta posibilidad, para facilitar el manejo posterior de los RDDs se recomienda que, salvo causa muy justificada, todos los registros sean del mismo tipo. Los RDDs que contienen parejas `(clave, valor)` reciben el nombre de RDDs de parejas (*pair RDD*). Este tipo particular de RDD admite las mismas operaciones que los RDDs normales, pero además proporcionan operaciones adicionales que permiten procesar los distintos registros por el valor de su clave. Más adelante explicaremos algunas de estas operaciones particulares.

Por otro lado, los RDDs han sido diseñados desde el inicio para ser distribuidos, es decir, los registros que lo componen se repartirán entre los distintos equipos de un clúster. Para realizar esta distribución, los RDDs se dividen en particiones. Cada partición se almacena únicamente en un proceso *executor* dentro de un nodo del clúster, aunque un proceso *executor* puede albergar distintas particiones de distintos RDDs. El número de particiones en las que dividir un RDD se puede configurar e incluso cambiar a lo largo de la ejecución, aunque por defecto se establecerá como el número de núcleos de procesamiento disponibles en el clúster.

Para decidir qué registros forman parte de cada partición, Spark utiliza *particionadores*, que son funciones que toman un registro y devuelven el número de la partición a la que pertenecen. Spark utiliza particionadores por defecto, aunque para mejorar el rendimiento del sistema se pueden definir particionadores

personalizados. La figura 6-4 muestra cómo se podría particionar un RDD de 13 parejas `(int, str)` sobre 3 procesos *executor* utilizando el rango de valores del primer elemento de la pareja.

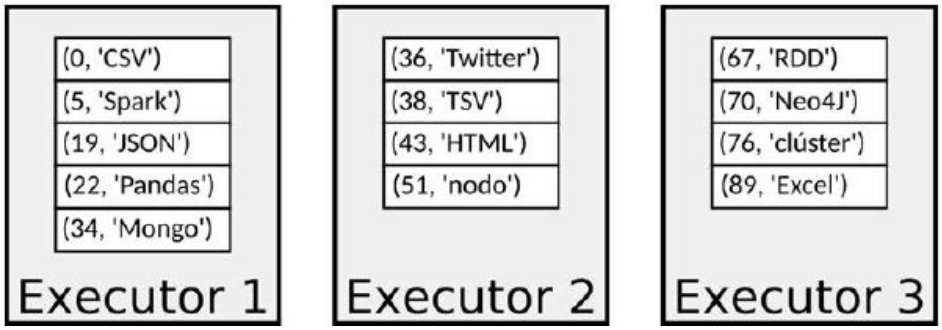


Figura 6-4. Particionado de un RDD por rangos de valores.

Otra de las características básicas de los RDDs es que son inmutables, es decir, no se pueden modificar ni actualizar. Una vez creado un RDD, este permanece inalterable durante toda la ejecución del programa. Los RDDs admiten dos tipos de operaciones, *transformaciones* y *acciones*, pero ninguna de ellas modifica el RDD. Las transformaciones son operaciones que toman un RDD de partida y crean un nuevo RDD, dejando el original intacto. Ejemplos de transformaciones son aplicar una función a todos los registros del RDD (por ejemplo, sumar una cierta cantidad), filtrar únicamente aquellos registros que cumplan una cierta condición u ordenarlos mediante algún campo. Por otro lado, las acciones son operaciones que realizan algún cómputo sobre el RDD y devuelven un valor, dejando también el RDD original inalterado. Ejemplos de acciones son sumar todos los elementos almacenados en un RDD de números, generando un valor final, o volcar un RDD a un fichero de texto. En los siguientes apartados veremos distintos ejemplos de transformaciones y acciones y cómo ejecutarlas en Spark desde Python.

Por último, los RDDs destacan por su resiliencia, es decir, su capacidad de recuperar su estado inicial cuando hay algún problema. Como todos los sistemas distribuidos, Spark debe manejar situaciones en las que algún equipo del clúster deja de responder ya sea por errores internos (fallo de alimentación, reinicio, error en el disco duro), como por problemas de conexión. Recordemos que los RDDs han sido divididos en particiones, y cada partición ha sido almacenada en un proceso *executor*, así que todas las particiones almacenadas en un equipo que está fallando dejarán de ser accesibles, presentando una potencial pérdida de datos. Como los RDDs son inmutables y se crean únicamente a partir de transformaciones de RDDs anteriores, Spark garantiza la disponibilidad de datos, repitiendo todas las transformaciones programadas desde el último RDD que esté disponible. De esta manera puede regenerar las particiones perdidas. Esto se logra a cambio de tiempo de cómputo,

pero esto permite continuar con el proceso en lugar de finalizar con un mensaje de error o esperar a que el equipo que está fallando vuelva a estar disponible. Lo más interesante de esta resiliencia es que es totalmente transparente al programador, ya que todo el proceso de regeneración de RDDs lo realiza Spark en segundo plano en caso de ser necesario.

A continuación, describiremos la creación de RDDs, proceso en el que juega un papel fundamental el mencionado `SparkContext`. También veremos en detalle cómo aplicar transformaciones y acciones sobre los RDDs, incluyendo transformaciones que únicamente se pueden aplicar sobre RDDs de parejas. Todos los fragmentos de código mostrados se pueden ejecutar en el *notebook* del capítulo disponible en el repositorio de código del libro.

CREACIÓN DE RDDS

En el anterior apartado hemos explicado que los RDDs son inmutables y que se crean mediante transformaciones de otros RDDs. Sin embargo, antes de poder comenzar la secuencia de transformaciones necesitamos un RDD (o varios) con los valores iniciales que queremos procesar. Estos RDDs se crearán a partir de valores almacenados en memoria dentro del proceso *driver* o a partir de ficheros accesibles desde el clúster. En ambos casos se creará un RDD y sus distintas particiones se distribuirán entre los diferentes procesos *executor*.

Para poder crear un RDD inicial, será necesario invocar a métodos del `SparkContext` en nuestro programa Spark. Si se ha configurado Spark y Jupyter tal y como se explica en el apéndice XX, todos los *notebooks* que se abran tendrán definida por defecto una variable `sc` que apuntará al objeto `SparkContext` del clúster actual. Según las instrucciones del apéndice XX, la invocación a `pyspark` lanzará Spark en modo local con 2 procesos *executor*.

La manera más sencilla de construir un RDD es partir de una colección de objetos en la memoria del proceso *driver*. Para ello, se hace uso del método `parallelize` del objeto `sc`:

```
>>> r = sc.parallelize([1, 2, 3, 4, 5])
```

Esta línea crea un RDD de 5 registros con los números del 1 al 5, y lo enlaza a la variable `r`. Si consultamos el tipo de esta variable mediante

```
>>> type(r)
```

comprobaremos que el tipo es el esperado `pyspark.rdd.RDD`. Debido al sistema de tipos de Python, el sistema Spark no tiene más información sobre los elementos que contiene el RDD, únicamente que se trata de un RDD. De la misma manera podríamos crear un RDD de 3 cadenas de texto:

```
>>> r = sc.parallelize(["hola", "hi", "ciao"])
```

Podríamos incluso utilizar funciones Python que devuelvan colecciones para construir RDDs más complejos o extensos. Por ejemplo, podíamos construir un RDD con los primeros 100 cuadrados perfectos usando comprensiones de listas:

```
>>> r = sc.parallelize([i*i for i in range(1,101)])
```

Al crear RDDs a partir de colecciones en memoria del proceso *driver* estamos limitados a la memoria que este proceso tenga disponible. Por eso, la manera más común de generar nuestros RDDs iniciales será cargarlos desde ficheros. Concretamente para el caso de ficheros de texto Spark proporciona el método `textFile`, que carga un fichero de texto y genera un RDD de cadenas de texto con un registro por cada línea. Por ejemplo, si tenemos un fichero de texto de 8 líneas en la ruta `data/file.txt` del sistema de ficheros local lo cargaríamos mediante:

```
>>> r = sc.textFile("data/Cap6/file.txt")
```

El RDD resultante tendría 8 registros, cada uno con una línea del fichero. En el caso de ejecución en modo local, la ruta será relativa al directorio desde donde hemos lanzado `pyspark`. En modo distribuido deberíamos establecer la ruta completa con el protocolo correspondiente al sistema de ficheros local, y además dicho fichero deberá existir en todos los equipos del clúster:

```
>>> r = sc.textFile("file:///data/Cap6/file.txt")
```

No obstante, `textFile` también puede abrir archivos de texto en sistemas de ficheros distribuidos como HDFS o en Amazon S3. En ese caso habría que utilizar el protocolo adecuado:

```
>>> r = sc.textFile("hdfs://node:port/data/file.txt")
>>> r = sc.textFile("s3n://bucket/file.txt")
```

Otra funcionalidad de `textFile` es que nos permite cargar varios ficheros a la vez usando rutas con comodines, o rutas que apunten a directorios:

```
>>> r = sc.textFile("data/Cap6/*.txt")
>>> r = sc.textFile("data/Cap6/")
```

En el primer caso, Spark leerá todos los ficheros que encajen con el nombre, y creará un RDD en el que cada registro es una línea de los ficheros. En el segundo caso, Spark actuará de forma similar, pero procesando todos los ficheros de texto que existan en el directorio, independientemente de su nombre o extensión. Sin embargo, al cargar varios ficheros a la vez dejaremos de conocer de qué fichero procede cada línea. En general esta información no importará, pero en algunos casos concretos querremos conservarla y utilizarla en el futuro. Para conseguirlo utilizaremos el método `wholeTextFiles`, que recibe una ruta y devuelve un RDD de parejas `(str, str)`: el primer elemento de cada registro será la ruta del fichero, y el segundo elemento será el texto completo del fichero, incluyendo saltos de línea. Por ejemplo, si tenemos 2 ficheros de texto en el directorio `data`, la siguiente instrucción generará un RDD de dos registros, uno por cada fichero:

```
r = sc.wholeTextFiles("data/Cap6/*.txt")
```

Además de los métodos presentados, Spark también dispone de métodos para crear RDDs a partir de ficheros en formato Hadoop (`sc.hadoopFile` y `sc.newAPIHadoopFile`) o ficheros con objetos serializados con *Pickle* (`sc.pickleFile`).

ACCIONES

Como ya hemos comentado, las acciones son operaciones que realizan algún cómputo sobre todo un RDD y devuelven un valor, dejando el RDD original en el mismo estado (pues es inmutable). Es importante darse cuenta de que un RDD puede almacenar gigabytes, terabytes o incluso petabytes de datos de manera distribuida entre distintos nodos del clúster, pero el valor generado por las acciones se enviará al proceso *driver*. Por lo tanto, antes de lanzar una acción es importante garantizar que el valor resultante puede ser almacenado en la memoria del proceso *driver*. Una excepción a esta regla son las acciones que vuelcan RDDs a ficheros, ya que estos métodos no devolverán ningún valor. Una característica de las acciones es que son *impacientes*, es decir, al lanzar una acción sobre un RDD esta comenzará inmediatamente su ejecución de manera distribuida a lo largo del clúster. Esto contrasta con la *pereza* de las transformaciones, que veremos más adelante.

collect, take y count

Cuando estamos escribiendo un programa Spark de manera interactiva en Jupyter, lo más normal es que hayamos creado decenas de RDDs mediante diferentes transformaciones. En estos casos, es probable que no recordemos lo que almacena cada uno. De la misma manera, mientras estamos realizando pruebas con distintas transformaciones necesitamos visualizar el RDD resultante para confirmar que contiene los valores esperados. Para estas situaciones Spark nos proporciona 2 métodos: `collect` y `take`. Ambos métodos recorren el RDD y devuelven una lista Python con todos sus elementos. Por ejemplo:

```
>>> r = sc.parallelize([1,2,3,4,5,6])
>>> r.collect()
[1, 2, 3, 4, 5, 6]
```

Este código genera un RDD `r` de 6 enteros en el clúster y mediante el método `collect` recupera todos los registros y los devuelve al proceso *driver*, generando la lista *[1, 2, 3, 4, 5, 6]*. Este método es muy útil para RDDs pequeños, pues nos permite de manera cómoda ver todos sus elementos, pero para RDDs grandes presenta dos problemas. Por un lado, la salida será difícil de leer y habrá que desplazarse entre una larga lista de elementos. Pero, más importante, se generará una gran transferencia de datos desde los nodos del clúster al proceso *driver*. Esto posiblemente congestione el proceso *driver*, que puede llegar a fallar si no tiene suficiente memoria disponible. Para casos en los que los que queremos consultar los elementos de un RDD muy grande sin colapsar el proceso *driver* podemos usar el método `take`, que devuelve los primeros elementos del RDD:

```
>>> r = sc.parallelize(range(1000))
>>> r.take(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

El método `take` recibe como parámetro el número *n* de elementos a devolver, y devuelve al proceso *driver* una lista con los primeros *n* elementos del RDD. En el código anterior, `r` es un RDD de 1000 elementos, pero gracias a `take` devolvemos únicamente los 10 primeros. Además de `take`, existen dos métodos relacionados para obtener elementos de un RDD: `takeOrdered` y `takeSample`. Ambos reciben el número de elementos a devolver, pero permiten tener en cuenta un orden concreto de los elementos o considerar un muestreo aleatorio de los elementos a devolver. Se pueden encontrar más detalles en la documentación de `pyspark.RDD` (ver apéndice).

Por último, el método `count` calcula el número de elementos de un RDD y lo devuelve como un entero. Por ejemplo, el siguiente código crea un RDD de 1000 elementos enteros y calcula su tamaño:

```
>>> r = sc.parallelize(range(1000))
>>> r.count()
1000
```

reduce y aggregate

Una de las acciones más útiles sobre RDDs es `reduce`, que nos permite recorrer todos los valores de un RDD y calcular un valor en relación con ellos, es decir, reduce un RDD a un único valor. La manera de calcular este valor es totalmente adaptable, ya que la función de reducción `f` aplicada será el parámetro que pasaremos. Dado un RDD con elementos de tipo `T`, la función de reducción es una función binaria que acepta dos elementos de tipo `T` y devuelve un valor del mismo tipo `T`, es decir, tiene tipo `T` *x* `T` → `T`. Se puede pensar en `reduce(f)` como un método que aplica la función de reducción `f` a los 2 primeros elementos, luego aplica de nuevo la función *f* al valor resultante y al tercer elemento, y así sucesivamente hasta que procesa el último elemento y produce el valor final. La figura 6-5 muestra este proceso para calcular la suma de un RDD de 5 elementos enteros.

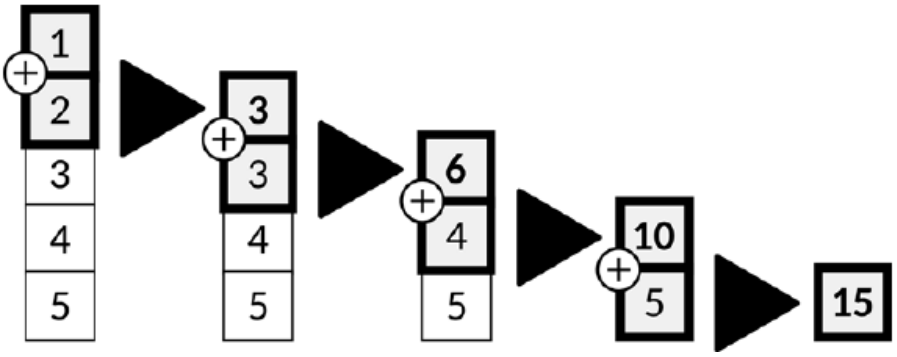


Figura 6-5. Reducción mediante suma sobre 5 elementos.

Para crear un RDD con los números del 1 al 5 y calcular su suma, definiríamos una función de reducción `add` e invocaríamos a `reduce` como sigue:

```
def add(x, y):
   return x + y

>>> r = sc.parallelize(range(1,6))
>>> r.reduce(add)
15
```

El resultado es 1 + 2 + 3 + 4 + 5 = 15, tal como esperamos. Es este caso hemos utilizado una función add definida por nosotros, pero podríamos haber utilizado una función anónima definida directamente en la invocación:

```
>>> r.reduce(lambda x,y: x+y)
15
```

La función de reducción puede ser tan compleja como necesitemos. Por ejemplo, dado un RDD de números podríamos multiplicar únicamente los números positivos que aparecen en un RDD:

```
def multiply_positive(x, y):
   if x > 0 and y > 0:
      return x*y
   elif x > 0:
      return x
   elif y > 0:
      return y
   else:
      return 1 # Elemento neutro

>>> r = sc.parallelize([-1, 2, 1, -5, 8])
>>> r.reduce(multiplica_positivos)
16
```

En este caso el resultado es 16 = 2 * 1 * 8. La función de reducción realiza la distinción de casos: si ambos parámetros son positivos, devuelve su multiplicación; si únicamente uno de esos parámetros es positivo, lo devuelve directamente; en otro caso devuelve 1 como elemento neutro de la multiplicación.

Es importante darse cuenta de que reduce lanzará una excepción si el RDD origen está vacío, mientras que en RDDs de un único elemento no aplicará la función de reducción y devolverá directamente dicho elemento:

```
>>> r = sc.parallelize([])
>>> r.reduce(lambda x,y : x+y)
---------------------------------
ValueError
(...)
ValueError: Can not reduce() empty RDD

>>> r = sc.parallelize([1])
>>> r.reduce(lambda x,y : x+y)
1
```

Otro aspecto importante del método `reduce` es que, dado que el RDD está particionado en distintos fragmentos, y cada uno está almacenado en un proceso *executor* diferente, no está garantizado el orden en el que se invocará a la función de reducción. Por ello, para garantizar la unicidad del resultado, es imprescindible que la función de reducción *f* sea conmutativa y asociativa, es decir, que para dos valores cualquiera *x,y, f(x,y) = f(y,x),* y que para cualquier tríada *x,y,z, f(x,f(y,z)) = f(f(x,y),z)*. De esta manera, Spark podrá distribuir los cómputos intermedios entre los procesos *executor* de la manera que sea más eficiente en cada momento y finalmente combinar los resultados parciales. Ejemplos de funciones conmutativas y asociativas son la suma, la multiplicación, el mínimo o el máximo. Sin embargo, funciones como la resta o la división no son conmutativas ni asociativas y por tanto no se pueden usar para reducir. Un ejemplo claro de este problema se da con la resta si variamos el número de particiones de un RDD:

```
>>> r = sc.parallelize(range(3), 1)
>>> r.reduce(lambda x,y: x-y)
-3
>>> r = sc.parallelize(range(3), 2)
>>> r.reduce(lambda x,y: x-y)
1
```

En ambos casos el RDD contiene los elementos 0, 1, 2. En el primer caso configuramos el RDD para tener una única partición. En este caso el resultado es el esperado si se procesan los elementos en orden: -3 = (0 - 1) - 2. Sin embargo, al dividir el RDD en dos particiones el resultado es 1. En este caso Spark ha creado las particiones [0] y [1, 2]. Al aplicar la función de reducción en cada partición se obtiene 0 y -1 = 1 - 2, respectivamente. Al combinar estos resultados parciales de nuevo mediante la función de reducción se produce el resultado inesperado final 1 = 0 - (-1).

En el método `reduce` la función de reducción está forzada a devolver un valor del mismo tipo que los elementos almacenados en el RDD. En algunos casos nuestra función de reducción encajará fácilmente en esta restricción, pero en otros será imposible.

Pensemos en un RDD de cadenas de texto, donde queremos contar el número total de caracteres `'h'` que aparecen. El resultado esperado es un número entero, así que la función de reducción debería devolver enteros. Esto obliga a dicha función a tomar dos argumentos enteros. Pero esto es incompatible con nuestro RDD, ya que, al partir de un fichero de cadenas de caracteres (tipo `str`), debería tomar dos argumentos de tipo `str` y devolver un valor `str`. En este caso no podríamos usar el método `reduce`, pero afortunadamente Spark nos proporciona una acción similar a `reduce` pero más flexible: `aggregate`. Este método acepta 3 parámetros:

- Un valor inicial `zeroValue` para el acumulador, que tendrá tipo `C`.
- Una función `seqOp` para combinar elementos de nuestro RDD (de tipo `T`) con el acumulador de tipo `C`, devolviendo un valor de tipo `C` (`C` *x* `T` → `C`).
- Una función `combOp` para combinar dos acumuladores de tipo `C` y devolver un valor de tipo `C`.

El resultado final será un valor de tipo `C`, como los acumuladores. Con esta acción ya podemos procesar el RDD y contar el número de veces que aparece el carácter `'h'`. Para ello:

- Usamos 0 como valor inicial para el acumulador, de tipo `int`.
- La función `seqOp` toma un acumulador de tipo `int` con el número de `'h'` encontradas hasta el momento y una cadena de texto. Debe devolver un valor de tipo `int` con el contador actualizado sumando las `'h'` que aparecen en la cadena actual.
- Por último, la función `combOp` recibe dos acumuladores y los combina. En este caso se trata simplemente de la suma de enteros.

El siguiente fragmento de código crea un RDD de 3 cadenas de texto y aplica `aggregate` para contar el número de caracteres `'h'` que aparecen. Como se puede ver, hemos optado por escribir las funciones de manera anónima, pero se podrían haber definido previamente y pasar el nombre de función como argumento:

```
>>> r = sc.parallelize(["hola", "hi", "ciao"])
>>> r.aggregate(0, lambda c, s : c + s.count('h'),
                lambda c1, c2: c1 + c2)
2
```

En este caso el resultado es 2: una aparición en la cadena `"hola"` y otra en `"hi"`. Para contar el número de apariciones del carácter `'h'` en una cadena `s` hemos utilizado directamente el método `s.count('h')`. Este código distingue entre mayúsculas y minúsculas. Si quisiéramos ignorar este aspecto podríamos refinar la segunda función para que transforme la cadena de texto a minúsculas antes de realizar el recuento.

Salvar RDDs en ficheros

Otro tipo de acciones relevantes sobre RDDs, son las relacionadas con su volcado a ficheros. En este apartado únicamente nos centraremos en la grabación en ficheros de texto, pero Spark permite salvar RDDs en otros formatos adecuados en el ecosistema Hadoop como *sequence files*. Recomendamos consultar la

documentación de la clase `pyspark.RDD` relativa a los métodos `saveAsSequenceFile`, `saveAsNewAPIHadoopDataset` o `saveAsNewAPIHadoopFile` para más información sobre estas posibilidades.

Para la grabación de un RDD como fichero de texto, Spark proporciona el método `saveAsTextFile`, que permite volcar cada elemento de un RDD a una línea de texto. Este método recibe como parámetro la ruta al directorio donde se almacenarán los distintos ficheros resultantes del RDD. Es importante observar que `saveAsTextFile` genera varios ficheros debido a su naturaleza distribuida. Como un RDD estará particionado entre varios procesos *executor*, al volcar un RDD a texto cada partición generará un fichero diferente. Cada fichero tendrá un nombre `part-XXXXX`, donde `XXXXX` es una numeración correlativa. Además de estos ficheros, Spark creará un fichero vacío de nombre `_SUCCESS` indicando que el proceso de grabación ha terminado con éxito. Al igual que la lectura de ficheros de texto, `saveAsTextFile` admite distintos protocolos como el sistema de ficheros local (`file://`), HDFS (`hdfs://`) o S3 (`s3n://`).

El siguiente ejemplo crea un RDD de enteros del 0 al 999 almacenado en 2 particiones y lo almacena en el directorio local */data/nums*:

```
>>> r = sc.parallelize(range(1000), 2)
>>> r.saveAsTextFile("file:///data/nums")
```

Tras estas instrucciones se habrá creado una carpeta `/data/nums` con 3 ficheros: `part-00000` con los números del 0 al 499, uno en cada línea; `part-00001` con los números del 500 al 999, uno en cada línea; y `_SUCCESS` indicando que la grabación ha tenido éxito. Incluso en el caso de que el RDD tenga una única partición, `saveAsTextFile` creará un directorio con un único fichero `part-00000` además del fichero `_SUCCESS`.

También hay que tener en cuenta que la ruta pasada como parámetro no debe existir en el sistema de ficheros, ya que Spark trata de crear una carpeta nueva con ese nombre. Si por error invocamos a `saveAsTextFile` con una ruta existente, nos devolverá una excepción como la siguiente:

```
>>> r.saveAsTextFile("file:///data/nums")
----------------------------------------
Py4JJavaError Traceback (most recent call last)
(...)
Output directory file:/tmp/nums already exists
(...)
```


TRANSFORMACIONES

Las transformaciones son las operaciones de los RDDs complementarias a las acciones, y se caracterizan por tomar un RDD de origen y generar otro RDD como resultado. El RDD resultante puede almacenar datos del mismo tipo, aunque lo más común es que genere un RDD con elementos de tipo diferente al origen. Otra característica relevante de las transformaciones es que, en contraposición a las acciones, son perezosas. Eso quiere decir que al ejecutar una transformación Spark *no realizará ningún cómputo real* con el RDD, únicamente anotará los datos de la transformación para ejecutarla en el futuro. Esta pereza permite que Spark agrupe varias transformaciones consecutivas y las ejecute de manera agrupada en lo que se conoce como una etapa (*stage*) de cómputo. Es por ello que al lanzar una transformación, Spark terminará de inmediato, pero al ejecutar una acción (por ejemplo `collect` o `count`) comenzará a ejecutar en orden todas las transformaciones pendientes y eso requerirá un tiempo de cómputo. Por tanto, las acciones *disparan* la ejecución de las transformaciones, pero solo hasta el punto necesario para devolver el valor requerido por la acción.

A continuación, veremos algunas de las transformaciones más importantes, aunque recomendamos al lector consultar la documentación de `pyspark.RDD` para conocer el resto de transformaciones disponibles.

map y flatMap

La transformación `map` permite aplicar una función elemento a elemento a un RDD. Si tenemos un RDD con elementos de tipo `T` y una función `f` que acepta elementos de tipo `T` y produce valores de tipo `V`, la transformación `map(f)` generará un RDD de elementos de tipo `V`. El resultado será por tanto un RDD con exactamente el mismo número de elementos que el RDD original, donde cada elemento es el resultado de aplicar la función `f` a un elemento del RDD original.

Un ejemplo sencillo de la transformación es incrementar en uno todos los números almacenados en un RDD:

```
>>> r = sc.parallelize([1,2,3,4])
>>> r2 = r.map(lambda x: x + 1)
>>> r2.collect()
[2, 3, 4, 5]
```

A partir del RDD `r` con los elementos `[1,2,3,4]` creamos un nuevo RDD `r2` sumando 1 a cada elemento, obteniendo por tanto `[2,3,4,5]`. Como la función de incremento es muy sencilla, hemos optado por definirla como función anónima

directamente en la invocación (`lambda x: x + 1`), pero al igual que con `reduce` o `aggregate` podríamos haber podido definir dicha función previamente:

```
def increment(x):
   return x + 1

>>> r = sc.parallelize([1,2,3,4])
>>> r2 = r.map(increment)
>>> r2.collect()
[2, 3, 4, 5]
```

Este ejemplo es sencillo, porque la función devuelve valores del mismo tipo que su parámetro, pero en general podemos devolver cualquier tipo diferente. Por ejemplo, a partir de un RDD de cadenas de texto podemos obtener un RDD con sus longitudes:

```
>>> r = sc.parallelize(["hola","hi","ciao"])
>>> r2 = r.map(lambda x: len(x))
>>> r2.collect()
[4, 2, 4]
```

El resultado es un RDD con la longitud del primer elemento `"hola"` (4), la longitud del segundo elemento `"hi"` (2) y la longitud del tercer elemento `"ciao"` (4).

El resultado de la función aplicada por `map` puede ser de un tipo compuesto como tuplas o listas. Por ejemplo, si tenemos un RDD con cadenas de texto en formato CSV, podríamos utilizar la transformación `map` para obtener un RDD de listas almacenando los elementos por separado. Para analizar cadenas de texto en formato CSV, usaremos la biblioteca `csv` como vimos en el capítulo 1. Concretamente usaremos `csv.reader` pasando como parámetro una lista unitaria con una única cadena de texto.

```
>>> import csv
>>> r = sc.parallelize(["1,5,7","8,2,4"])
>>> r2 = r.map(lambda x: list(csv.reader([x]))[0])
>>> r2.collect()
[['1', '5', '7'], ['8', '2', '4']]
```

El resultado es un RDD con dos elementos, cada uno de tipo lista de cadenas de texto. A su vez, cada lista contiene 3 elementos, tal y como aparecían en el RDD original. Dado que `csv.reader` necesita que su parámetro sea un objeto iterable, pasamos la cadena de texto `x` dentro de una lista unitaria. Esto hace que el resultado

de `list(csv.reader([x]))` sea una lista unitaria con una lista dentro que contiene el resultado de trocear la cadena, y por eso extraemos su primer (y único) elemento con el operador `[0]`. Es importante darse cuenta de que el RDD resultante almacena listas de cadenas de texto y no números. Si quisiéramos obtener listas de números, tendríamos que ampliar la función que pasamos a la transformación `map` con una fase más o, mejor aún, establecer una segunda transformación que convierta listas de cadenas en listas de enteros. Para transformar una lista de cadenas en una lista de enteros podemos utilizar la comprensión de listas `[int(e) for e in l]`:

```
>>> r3 = r2.map(lambda l: [int(e) for e in l])
>>> r3.collect()
[[1, 5, 7], [8, 2, 4]]
```

Otra transformación similar a `map` es `flatMap`. Esta transformación acepta una función que toma un elemento del RDD y genera una lista (o algún dato que pueda ser recorrido) y la aplica a todos los elementos del RDD. A diferencia de `map`, `flatMap` aplana todas las listas obtenidas en el RDD. Es decir, si un elemento del RDD original genera la lista `[1, 2, 3]`, el aplanamiento conseguirá que el RDD tenga un elemento 1, seguido de un elemento 2 y un elemento 3. Dicho de otro modo, elimina las listas y se queda únicamente con sus elementos.

En el ejemplo en formato CSV anterior, el resultado de aplicar `flatMap` sería el siguiente:

```
>>> r = sc.parallelize(["1,5,7","8,2,4"])
>>> r2 = r.flatMap(lambda s: list(csv.reader([s]))[0])
>>> r2.collect()
['1', '5', '7', '8', '2', '4']
```

En este código, partimos de un RDD de dos elementos. Como cada elemento genera una lista de 3 elementos, el RDD resultante tiene 6 elementos: los 3 elementos de la primera lista seguidos de los 3 elementos de la segunda.

filter

La transformación `filter` permite seleccionar de un RDD únicamente aquellos elementos que cumplan una determinada condición. De esta manera, el RDD resultante contendrá un subconjunto de los elementos del RDD original. El método `filter` recibe como parámetro una función que calcula la condición deseada. Si el RDD contiene elementos de tipo `T`, la función debe aceptar elementos de tipo `T` y

devolver un booleano. Por ejemplo, podemos usar `filter` para filtrar un RDD de números del 2 al 30 y quedarnos únicamente con los números primos. Para ello definiremos una función `es_primo` que recibe un número y comprueba si es primo o no considerando todos sus posibles divisores a partir del 2:

```
def es_primo(x):
   for i in range(2,x):
      if x % i == 0:
         return False
   return True

>>> r = sc.parallelize(range(2,31))
>>> r2 = r.filter(es_primo)
>>> r2.collect()
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

En este caso hemos pasado la función usando su nombre, pero podíamos haber creado la condición directamente en la propia invocación mediante una función anónima.

RDDs de parejas

Los RDDs de parejas son aquellos que contienen parejas *(clave, valor)*. Este tipo de RDDs son importantes, ya que Spark nos va a permitir aplicar algunas transformaciones particulares que tienen en cuenta la clave de cada registro.

Una de estas transformaciones específicas es `mapValues`, que nos permite aplicar una función a todos los registros pero afectando, únicamente, a los valores de las parejas, dejando por tanto las claves inalteradas. Por ejemplo, podríamos incrementar todos los valores de un RDD de parejas de la siguiente manera:

```
>>> r = sc.parallelize([('a',0), ('b', 1),('c',2)])
>>> r2 = r.mapValues(lambda x: x+1)
>>> r2.collect()
[('a', 1), ('b', 2), ('c', 3)]
```

La transformación `mapValues` funciona de la misma manera que la transformación `map`, de hecho sería muy sencillo expresar el mismo cómputo utilizando `map`. Lo único que tendríamos que hacer es utilizar una función que acepte una pareja, deje el primer elemento (la clave) igual y transforme el segundo elemento (el valor). Por tanto, el ejemplo anterior se podría escribir como:

```
>>> r = sc.parallelize([('a',0), ('b', 1),('c',2)])
>>> r2 = r.map(lambda p: (p[0], p[1]+1))
>>> r2.collect()
[('a', 1), ('b', 2), ('c', 3)]
```

Lo único que hemos cambiado es la función pasada como parámetro. Ahora acepta una pareja `p` y devuelve una pareja con la misma clave (`p[0]`) y como valor el resultado de incrementar el valor anterior (`p[1]+1`).

Al igual que `mapValues`, existe una transformación `flatMapValues` similar a `flatMap` pero adaptada a RDDs de parejas. Dado un RDD de parejas con tipo `(K,V)` esta transformación recibe una función que acepta un valor de tipo `V` y genera una lista de algún tipo `V2`. Esta función se aplicará al valor de cada registro. Por cada uno de estos elementos de tipo `V2` de la lista resultante, `flatMapValues` generará un registro contiendo la clave original y dicho elemento. Por ejemplo, consideremos un RDD de parejas con tipo `(str,str)`, donde el valor es una cadena con números en formato CSV. Si aplicamos `flatMapValues` usando la función que vimos en el ejemplo de `map` para partir la cadena CSV el resultado sería el siguiente:

```
>>> import csv
>>> r = sc.parallelize([('a','1,5,7'),('b','8,2')])
>>> r2 = r.flatMapValues(lambda x: list(csv.reader([x]))[0])
>>> r2.collect()
[('a','1'),('a','5'),('a','7'),('b','8'),('b','2')]
```

Como se puede ver, el primer registro ha creado los 3 elementos `('a','1')`, `('a','5')` y `('a','7')`; mientras que el segundo elemento ha creado los dos registros `('b','8')` y `('b','2')`. En ambos casos, cada registro ha creado un nuevo elemento por cada valor de la lista resultante de dividir la cadena CSV.

Otra de las transformaciones particulares de los RDDs de parejas es `groupByKey`, que como su nombre indica agrupará todos los registros con el mismo valor de clave en un único registro. Este registro unificado tendrá como clave la misma de los elementos agrupados y como valor un objeto iterable de la clase `pyspark.resultiterable.ResultIterable` con los valores de todos los registros que compartían dicha clave. Es importante darse cuenta de que este objeto no es una lista de Python propiamente dicha, es decir, no tiene tipo `list`. Esto quiere decir que si lo mostramos por pantalla no veremos sus elementos sino algo parecido a:

```
<ResultIterable at 0x7f25cc980f98>
```

Sin embargo, al tratarse de un objeto iterable podremos recorrerlo de la misma manera que las listas de Python. Por ejemplo, para agrupar los registros con el mismo valor de clave de un RDD ejecutaríamos:

```
>>> r = sc.parallelize([('a',3.14), ('b',9.4), ('a',2.7)])
>>> r2 = r.groupByKey()
>>> r2.collect()
[('a', <ResultIterable at 0x7f25cc980f98>),
 ('b', <ResultIterable at 0x7f25cc9808d0>)]
```

Si quisiéramos visualizar los elementos que hay en cada uno de sus iterables podríamos aplicar una transformación `mapValues` para transformar todos estos iterables en listas Python, que se mostrarían sin problema:

```
>>> r3 = r2.mapValues(lambda x: list(x))
>>> r3.collect()
[('a', [3.14, 2.7]), ('b', [9.4])]
```

Spark también nos proporciona la transformación `reduceByKey` para aplicar una función de combinación a todos los valores asociados a una misma clave. Se puede pensar que esta transformación primero agrupa todos los valores asociados a la misma clave, y luego recorre esta colección combinando los valores mediante la función de reducción. La función de reducción debe ser asociativa y conmutativa, al igual que con la acción `reduce`. Si tomamos un RDD de parejas de tipo `(K,V)` como origen, deberemos aplicar una función de reducción binaria que tome como parámetros dos elementos de tipo `V` y genere un resultado de tipo `V`. El resultado será otro RDD de parejas del mismo tipo `(K,V)`.

Un ejemplo sencillo de `reduceByKey` sería sumar todos los valores asociados a la misma clave:

```
>>> r = sc.parallelize([('a',2), ('b', 1),('a',3)])
>>> r2 = r.reduceByKey(lambda x,y: x+y)
>>> r2.collect()
[('a', 5), ('b', 1)]
```

El comportamiento de `reduceByKey` se podría simular mediante dos transformaciones encadenadas: primero `groupByKey` para agrupar todos los valores asociados a una misma clave y luego `mapValues` (o `map` procesando parejas) para recorrer el iterable resultante y combinar sus elementos. Aunque desde el punto de vista de los valores finales el resultado es el mismo, se recomienda utilizar `reduceByKey` ya que el rendimiento será mejor.

Por último, mencionar que Spark proporciona `aggregateByKey`, similar a `reduceByKey` pero con la flexibilidad que daba `aggregate`: un valor inicial para el acumulador, una función para combinar un valor con el acumulador y una función para combinar dos acumuladores.

Transformaciones combinando dos RDDs

Hasta ahora hemos presentado transformaciones que únicamente afectaban a un RDD. Aceptaban una o más funciones como parámetros y las usaban para transformar el RDD en otro. Sin embargo, hay algunas transformaciones interesantes que combinan datos de dos RDDs. Entre ellas destacamos la unión, la intersección, la diferencia y los distintos tipos de combinaciones de RDDs (*joins*).

La unión (`union`) de dos RDDs genera un RDD con los elementos del primer RDD seguidos de los elementos del segundo RDD. Al contrario de lo que podía parecer por el nombre, no elimina elementos repetidos, es decir, estrictamente no es una unión de conjuntos, sino una concatenación de elementos. Un ejemplo:

```
>>> r1 = sc.parallelize([1,2,3,4])
>>> r2 = sc.parallelize([2,4,6])
>>> r3 = r1.union(r2)
>>> r3.collect()
[1, 2, 3, 4, 2, 4, 6]
```

Como el primer RDD tiene 4 elementos, y el segundo tiene 3 elementos, el RDD resultante tiene 7 elementos. Sin embargo, hay 2 elementos repetidos porque aparecen en ambos RDDs: el 2 y el 4. Si quisiéramos conseguir un RDD con elementos únicos deberíamos aplicar la transformación `distinct` al RDD resultante:

```
>>> r4 = r3.distinct()
>>> r4.collect()
[1, 2, 3, 4, 6]
```

La intersección (`intersection`) de RDDs funciona como la operación sobre conjuntos: elige únicamente aquellos elementos que aparecen en ambos RDDs. En este caso Spark sí que elimina todas las repeticiones que puedan aparecer, generando un RDD de elementos únicos. En el ejemplo siguiente el elemento 2 aparece repetido en el RDD `r2`, sin embargo, este elemento aparece una única vez en el RDD resultante:

```
>>> r1 = sc.parallelize([1,2,3,4])
>>> r2 = sc.parallelize([2,4,2,6])
>>> r3 = r1.intersection(r2)
>>> r3.collect()
[2, 4]
```

La diferencia (`subtract`) de RDDs selecciona aquellos elementos del primer RDD que no aparecen en el segundo RDD. Esta transformación es similar a la operación de diferencia de conjuntos, aunque si el primer RDD contiene elementos repetidos, estos aparecerían en el RDD resultante. En el siguiente ejemplo tenemos que `r1` contiene el elemento 1 duplicado. Como dicho elemento no aparece en `r2` entonces aparecerá en el RDD final dos veces, una por cada aparición en `r1`:

```
>>> r1 = sc.parallelize([1,2,3,4,1])
>>> r2 = sc.parallelize([2,6])
>>> r3 = r1.subtract(r2)
>>> r3.collect()
[1, 1, 3, 4]
```

Por último, Spark también proporciona distintos tipos de combinaciones (*joins*) de RDDs. Por ejemplo, podemos calcular el producto cartesiano (`cartesian`) de dos RDDs. Si el primer RDD tiene N elementos y el segundo M elementos, el resultado será un RDD con N x M parejas, una por cada posible combinación de un elemento del primer RDD con otro del segundo RDD. En el siguiente ejemplo tenemos un RDD de 3 elementos (1, 2 y 3) y otro de 2 elementos (`'a'` y `'b'`). Si calculamos el producto cartesiano produciremos 6 parejas: el primer elemento de `r1` combinado con los 2 elementos de `r2` `[(1,'a'), (1,'b')]`, el segundo elemento de `r1` con los 2 elementos de `r2` `[(2,'a'), (2,'b')]`, y lo mismo para el tercer elemento `[(3,'a'),(3,'b')]`. El resultado sería el siguiente:

```
>>> r1 = sc.parallelize([1,2,3])
>>> r2 = sc.parallelize(['a','b'])
>>> r3 = r1.cartesian(r2)
>>> r3.collect()
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

El resto de combinaciones de RDDs operan sobre RDDs de parejas, ya que necesitan una clave para encajar elementos del primer RDD con elementos del segundo RDD. Existen varios tipos de reuniones (`join`, `fullOuterjoin`, `leftOuterJoin` y `rightOuterJoin`) que se diferencian en el criterio para incluir o no los elementos. Por ejemplo, la transformación `join` combina dos RDDs de parejas seleccionando únicamente aquellas claves que aparecen en ambos RDDs (es decir, actúa como un *inner join*). En esos casos genera una pareja con la clave común a los

dos RDDs y como valor crea una pareja con el valor en el primer RDD seguido del valor en el segundo RDD. El siguiente ejemplo combina dos RDDs donde la única clave común es la `'b'`:

```
>>> r1 = sc.parallelize([('a',1),('b',2),('c',3)])
>>> r2 = sc.parallelize([('b',8),('d',7)])
>>> r3 = r1.join(r2)
>>> r3.collect()
[('b', (2, 8))]
```

Como se puede ver, se genera un RDD con un único elemento. Como clave tiene `'b'`, que aparecía en ambos RDDs, y como valor tiene la pareja `(2,8)`, donde el 2 proviene de `r1` y el 8 de `r2`. Si los RDDs implicados en la reunión contienen claves duplicadas, también se duplicarán en el RDD resultante. En el ejemplo siguiente tenemos que la clave `'b'` aparece en ambos RDDs, pero en el segundo RDD aparece dos veces: una vez asociado al valor 8 y otra vez asociado al valor 0. Al calcular la reunión, Spark combinará el valor de `'b'` en `r1` con ambos valores de `r2`, generando por tanto 2 elementos en el RDD resultado:

```
>>> r1 = sc.parallelize([('a',1),('b',2),('c',3)])
>>> r2 = sc.parallelize([('b',8),('d',7),('b',0)])
>>> r3 = r1.join(r2)
>>> r3.collect()
[('b', (2, 8)), ('b', (2, 0))]
```

EJEMPLO DE PROCESAMIENTO DE RDD

En las dos secciones anteriores hemos visto distintas acciones y transformaciones sobre RDDs, de manera aislada. Sin embargo, un programa Spark usual constará de distintas operaciones encadenadas para conseguir el objetivo deseado. En esta sección mostraremos un ejemplo combinando diferentes transformaciones y acciones para preprocesar el conjunto de datos original de los pasajeros del Titanic y conseguir una representación más fácil de utilizar en aprendizaje automático. Este preprocesado será una simplificación del aplicado en el capítulo anterior, ya que únicamente involucrará:

1. Cargar el fichero CSV original y almacenarlo en un RDD de diccionarios Python que relacionan el nombre de la columna con su valor.
2. Descartar las columnas *PassengerId*, *Name*, *Ticket* y *Cabin* y eliminar todas las entradas que tengan algún valor vacío en las columnas restantes .
3. Representar los valores de cada columna con su tipo adecuado (`int`, `float` o `str`).

El primer paso será cargar el fichero CSV como fichero de texto y dividir cada línea en una lista Python de 12 elementos, cada elemento refiriéndose a un atributo. Para ello utilizaremos el método `sc.textFile` para crear un RDD de cadenas de texto, luego dividiremos cada cadena en una lista usando la transformación `map` y finalmente utilizaremos la transformación `filter` para eliminar la cabecera del fichero CSV (que será un elemento más del RDD):

```
>>> import csv
>>> raw = ( sc.textFile("data/titanic.csv")
>>>           .map(lambda s: list(csv.reader([s]))[0])
>>>           .filter(lambda l: l[0] != 'PassengerId')
>>>       )
>>> print(raw.count())
891
```

En lugar de dar un nombre a cada uno de los RDDs intermedios, hemos agrupado las tres operaciones consecutivas para formar un RDD llamado `raw` con los datos en bruto. Es importante darse cuenta de los paréntesis que hay encerrando todas las operaciones, ya que indican a Python que se trata de la misma expresión, aunque se extienda entre varias líneas. Si se eliminan los paréntesis Python mostraría un error de sintaxis, pues en la línea donde está definido `raw` no aparece ninguna expresión bien formada. Hemos seguido el estilo que se utiliza normalmente con operaciones Spark encadenadas para facilitar su lectura: cada operación aparece en una línea y todas aparecen con el mismo sangrado que la primera. Para dividir la cadena de texto usando las comas hemos usado la biblioteca `csv`, y para eliminar la cabecera hemos filtrado todas las listas cuyo primer elemento sea distinto de la cadena de texto `'PassengerId'`. Este filtrado únicamente eliminará la cabecera, ya que el resto de listas contendrá un número en su primer elemento. El resultado será un RDD con 891 registros, donde cada registro será una lista de 12 cadenas de texto.

A partir del RDD con los datos en bruto eliminaremos aquellas entradas a las que les falte algún valor, ya que queremos tratar únicamente con entradas completas. Además, transformaremos las listas en diccionarios para poder acceder más fácilmente a cada uno de los campos, además de convertir los valores numéricos a tipos `int` y `float` para poder operar con ellos a la hora de normalizar. Aprovecharemos esta etapa para eliminar las columnas que no aportan demasiada información, concretamente *PassengerId*, *Name*, *Ticket* y *Cabin*, que están en las columnas 0, 3, 8 y 10, respectivamente. Para realizar todas estas operaciones definiremos dos funciones auxiliares:

```
def complete(l):
   for i in [1,2,4,5,6,7,9,11]:
     if l[i] == '':
        return False
   return True

def proyect_and_parse(l):
   return {
           'Survived': int(l[1]),
           'Pclass':   int(l[2]),
           'Sex':      l[4],
           'Age':      float(l[5]),
           'SibSp':    int(l[6]),
           'Parch':    int(l[7]),
           'Fare':     float(l[9]),
           'Embarked': l[11]
          }
```

Utilizaremos la función `complete` con una transformación `filter` para descartar del RDD todas aquellas entradas que almacenan una cadena vacía (que indica valor vacío) en alguna de las columnas que nos interesan, es decir, las columnas en posiciones 1, 2, 4, 5, 6, 7, 9 y 11. La función `proyect_and_parse` toma una lista y realiza dos tareas: la transforma en un diccionario utilizando como claves los nombres de los atributos y considerando únicamente los atributos que nos interesan, y además convierte las cadenas de texto que representan valores numéricos a tipos `int` y `float`. Una vez tenemos estas funciones, obtener el RDD de diccionarios se reduce a concatenar dos transformaciones:

```
>>> non_null = (
>>>   raw.filter(complete)
>>>      .map(proyect_and_parse)
>>> )
>>> print(non_null.count())
712
```

A partir del RDD `raw` primero se eliminan las listas incompletas, y posteriormente se transforman las listas completas a diccionarios. El resultado es un RDD `non_null` con 712 registros, es decir, había 179 entradas incompletas en nuestro RDD con datos en bruto. Además, cada registro del RDD `non_null` es un diccionario con exactamente las 8 claves que hemos introducido en la función `proyect_and_parse`.

CONCLUSIONES

En este capítulo hemos presentado Apache Spark y hemos visto cómo realizar cómputos masivos de manera distribuida utilizando RDDs. Aunque no es terriblemente complicado, el manejo de RDD requiere que definamos una alta cantidad de funciones (anónimas o con nombre) para aplicar la mayoría de sus transformaciones, lo que produce un aumento de la complejidad. Concretamente, en el capítulo anterior vimos que realizar el preprocesado del conjunto de datos de los pasajeros del Titanic era bastante sencillo si usábamos los `DataFrames` de `pandas`. En la sección anterior hemos realizado preprocesado simplificado sobre RDDs, y el proceso ha resultado mucho más complejo. Es verdad que los `DataFrames` de `pandas` no pueden almacenar una gran cantidad de información, mientras que los RDDs que hemos creado pueden escalar en un clúster y almacenar hasta petabytes de información. El código que hemos escrito para preprocesar RDDs escalaría de manera transparente a cualquier tamaño soportado por el clúster, pero la complejidad de las operaciones es más elevada.

Conscientes de esta situación, Spark proporciona en su componente SparkSQL otro modelo de datos que simplifica las operaciones sobre los datos sin perder la capacidad de manejar grandes tamaños y su distribución a lo largo de un clúster. En el siguiente capítulo presentaremos este tipo de datos de Spark, llamado también `DataFrame` a similitud de Pandas, y veremos cómo nos permite realizar aprendizaje automático aprovechando los recursos de un clúster de manera transparente.

REFERENCIAS

- Learning Spark: Lightning-fast Data Analysis. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. O'Reilly, 2015.
- High Performance Spark: Best Practices for Scaling & Optimizing Apache Spark. Holden Karau, Rachel Warren. O'Reilly, 2017.
- Página principal de Apache Spark: `https://spark.apache.org/`
- Documentación de la clase RDD: `https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD`

7 SPARKSQL Y SPARKML

SPARKSQL

Los RDDs son un modelo de datos muy adecuado para Spark ya que fueron diseñados para ser divididos en fragmentos y distribuidos a través de un clúster. Además, gracias a su inmutabilidad es sencillo recuperar los datos si algún equipo del clúster deja funcionar, ya que únicamente hay que repetir todas las transformaciones que dieron lugar al fragmento perdido. Como hemos visto en el apartado anterior, las acciones y transformaciones se deben definir en el lenguaje de programación que estemos usando (Scala, Java, Python o R). Por ello, es imprescindible tener conocimientos de programación para poder realizar cualquier manipulación sobre RDDs.

Para tratar de simplificar el uso de Spark y ampliar el sistema a más usuarios, desde la versión 1.0 (año 2014) Spark incluyó el componente SparkSQL como parte de su sistema. Este componente proporciona un nuevo modelo de datos, los `DataFrames`, que son tablas formadas por filas y columnas etiquetadas con un nombre. En este sentido, los `DataFrames` de Spark son muy parecidos a los `DataFrames` de `pandas` y a las tablas de las bases de datos relacionales como Oracle o MySQL, consiguiendo que la curva de aprendizaje de nuevos usuarios se suavice bastante al utilizar básicamente los mismos conceptos. De hecho, los `DataFrames` permiten definir las operaciones a ejecutar utilizando el lenguaje de consultas SQL, el estándar para realizar consultas en bases de datos relacionales. Por todo esto, los `DataFrame` permiten definir las operaciones de una manera más sencilla que los RDDs.

Además de la sencillez, los `DataFrames` también persiguen mejorar el rendimiento. Internamente están construidos sobre los RDDs, así que heredan todas sus capacidades de distribución y de recuperación a fallos. Al igual que los RDDs los `DataFrames` son inmutables y sus operaciones son perezosas. Esto quiere decir que no se pueden actualizar celdas concretas de un `DataFrame` sino que es necesario transformarlo en su totalidad y obtener uno nuevo. Además, las operaciones no se lanzan en el momento de ejecutar la instrucción sino cuando se necesitan los datos, por ejemplo para mostrar un resumen por pantalla o para volcar el `DataFrame` al sistema de ficheros. A diferencia de los RDDs, los `DataFrames` poseen un esquema de datos asociado que indica qué columnas tiene cada entrada y qué tipo de datos contiene cada columna. Gracias a esta información y al hecho de que las operaciones son perezosas, el optimizador de consultas (llamado Catalyst) puede planificar (fusionar, reordenar, etc.) las distintas operaciones para obtener el mejor rendimiento. Además, Spark utiliza la información del esquema para almacenar de manera más eficiente los datos en el clúster.

Antes de continuar con las distintas operaciones disponibles sobre `DataFrames`, comentaremos que existe un tercer modelo de datos en Spark: los `DataSets`. Este nuevo modelo de datos fue introducido en versión 1.6 de Spark (año 2016) y se asentó definitivamente a partir de la versión 2.0 ese mismo año. Los `DataSets` proporcionan dos variantes: `DataSets` con tipos, que permiten detectar algunos errores en tiempo de compilación a cambio de perder flexibilidad, y `DataSets` sin tipos, similares a los `DataFrames`. En el caso de Python, como es un lenguaje sin tipado estático, únicamente existe la versión de `DataFrames` que trataremos en esta sección. Sin embargo, recomendamos a los lectores revisar la documentación del API de `DataSets` si van a usar Spark desde Scala o Java.

Creación de DataFrames

Los `DataFrames`, al igual que los RDDs, se pueden crear a partir de datos en la memoria del proceso *driver* o a partir de ficheros almacenados en el clúster. En el apartado anterior vimos que la creación de RDDs se invoca a partir de un objeto `sc` de tipo `SparkContext`, que era el punto de entrada para programas que usan RDDs. En este caso, la creación de `DataFrames` tomará como punto de entrada un objeto `spark` de tipo `SparkSession`. Si se han configurado Spark y Jupyter tal y como se explica en el apéndice todos los *notebooks* que se abran tendrán definida por defecto una variable `spark` que apuntará al objeto `SparkSession` del clúster actual.

DATAFRAMES DESDE VALORES Y RDDS

La manera más sencilla de crear un `DataFrame` es a partir de una lista Python almacenada en el propio proceso *driver*. Por ejemplo, podemos crear un `DataFrame` de 2 columnas invocando a `spark.createDataFrame`:

```
>>> l = [('a',3.14), ('b', 9.4), ('a',2.7)]
>>> df = spark.createDataFrame(l, ['id','value'])
```

La primera línea construye una lista `l` de 3 parejas, donde el primer elemento es una cadena de texto y el segundo un número decimal. Cada uno de estos elementos será una fila en el `DataFrame` que creemos. Al invocar a `spark.createDataFrame` pasamos como parámetros los datos a incluir (la lista `l`) y la cabecera, que es una lista de cadenas de texto con el nombre que queremos dar a cada una de las columnas. Si queremos ver el `DataFrame` resultante podemos invocar al método `show(n)`, que mostrará las primeras `n` filas del `DataFrame` de manera tabulada. Si omitimos el parámetro `n` se mostrarán por defecto las primeras 20 filas:

```
>>> df.show()
+---+-----+
| id|value|
+---+-----+
|  a| 3.14|
|  b|  9.4|
|  a|  2.7|
+---+-----+
```

Como se puede ver, el `DataFrame df` tiene 3 filas y dos columnas *id* y *value*. Además, durante la creación del `DataFrame` Spark ha realizado una tarea muy importante: inferir el esquema de los datos. Podemos ver qué esquema ha obtenido para nuestro `DataFrame` invocando a `printSchema`, que mostrará el esquema en forma de árbol en modo texto:

```
>>> df.printSchema()
root
 |-- id: string (nullable = true)
 |-- value: double (nullable = true)
```

La primera columna tiene nombre *id* y almacena valores de tipo `str`, mientras que la segunda columna tiene nombre *value* y almacena valores de tipo `double`. El valor *nullable* indica si esa columna puede contener valores nulos, es decir, `None`. Para la inferencia de los tipos de datos de cada columna Spark ha comprobado los valores

almacenados en todas las filas para verificar que son del mismo tipo. Para que la creación de un `DataFrame` a partir de una lista tenga éxito, la lista debe contener tuplas de la misma longitud y almacenar valores de los mismos tipos en las mismas posiciones. Si esta restricción no se cumple, la invocación a `spark.createDataFrame` lanzará una excepción:

```
>>> l = [('a',3.14), ('b',True)]
>>> df = spark.createDataFrame(l, ['id','value'])
(...)
TypeError: field value: Can not merge type
   <class 'pyspark.sql.types.DoubleType'> and
   <class 'pyspark.sql.types.BooleanType'>
```

En este caso hay una discrepancia entre el tipo `double` que tiene el valor `3.14` y el tipo `bool` que tiene el valor `True`. Como no es posible fusionar ambos tipos, la creación del `DataFrame` se aborta con una excepción `TypeError`.

El parámetro donde pasamos los nombres de las columnas admite más valores posibles. Si omitimos ese parámetro (es decir, si su valor es `None`) entonces Spark seguirá infiriendo el esquema de datos pero asignará a cada columna nombres numéricos consecutivos: *_1*, *_2*, *_3*, etc. Además, si pasamos una descripción del esquema de datos esperado, durante la creación del `DataFrame` Spark verificará que cada fila cumple el esquema en lugar de inferirlo. La descripción de los esquemas de datos se realiza mediante objetos del módulo `pyspark.sql.types`, que incluye tipos predeterminados que podemos combinar para crear esquemas complejos. Por ejemplo, contamos con `StringType` para cadenas de texto, `BooleanType` para booleanos, `DoubleType` y `FloatType` para números en coma flotante con precisión simple y doble, `IntegerType` y `LongType` para enteros de 32 y 64 bits, `ArrayType` para listas homogéneas, etc. Para definir el esquema de un `DataFrame` debemos crear un objeto de tipo `StructType` que contenga tantos objetos `StructField` como columnas vaya a tener nuestro `DataFrame`. En el caso del ejemplo anterior tendríamos dos objetos `StructField`: uno para la columna *id* que almacena cadenas de texto, y otro para la columna *value* que almacena números decimales. Para construir un objeto `StructField` tendremos que pasar 3 parámetros: el nombre de la columna, el tipo de dato almacenado y un booleano indicando si dicha columna puede o no contener valores nulos. El esquema de datos concreto se definiría como sigue:

```
schema = StructType([ StructField('id',   StringType(), True),
                      StructField('value', FloatType(), False)
                    ])
```

En este esquema hemos tomado dos decisiones que la inferencia automática de Spark no consideraba. Por un lado, hemos establecido que la columna *value* almacena números en coma flotante de precisión simple (en lugar de la precisión doble que infería Spark) y por otro lado hemos fijado que la columna *value* no puede almacenar valores nulos. Si utilizamos este esquema de datos a la hora de crear el `DataFrame`, Spark comprobará que este se cumple en cada fila y lanzará una excepción si alguna fila no lo verifica:

```
>>> l = [('a',3.14), ('b', 9.4), (None, 2.7)]
>>> df = spark.createDataFrame(l, schema)
>>> df.printSchema()
root
 |-- id: string (nullable = true)
 |-- value: float (nullable = false)
>>> l = [('a',3.14), ('b', 9.4), ('a', True)]
>>> df = spark.createDataFrame(l, schema)
TypeError: field value: FloatType can not
  accept object True in type <class 'bool'>
```

El primer caso verifica el esquema de datos aunque la tercera fila contiene None en su columna *id*, ya que dicha columna puede contener valores nulos (`nullable = true`). Sin embargo, el segundo caso lanza una excepción `TypeError` porque la tercera fila contiene un booleano en la columna *value*, y según el esquema únicamente podría contener números en coma flotante de precisión simple.

Hasta ahora hemos visto cómo crear `DataFrames` a partir de listas de parejas, pero la misma función nos servirá para crear `DataFrames` a partir de RDDs. En este caso también podemos pasar una lista de cadenas como cabecera, omitir dicha cabecera o pasar una descripción del esquema de datos. En los dos primeros casos Spark inferirá el esquema de datos a partir de los valores del RDD, mientras que en el último caso utilizará el esquema de datos proporcionado y verificará que todas las filas lo cumplen. Esto se puede comprobar en el siguiente fragmento de código, donde primero se crea un RDD de parejas y luego se genera un `DataFrame` a partir de él:

```
>>> r = sc.parallelize([('a',3.14), ('b', 9.4), ('a', 2.7)])
>>> df = spark.createDataFrame(r, ['id','value'])
>>> df.printSchema()
root
 |-- id: string (nullable = true)
 |-- value: double (nullable = true)
```

DATAFRAMES DESDE FICHEROS

El uso de ficheros en formato CSV o JSON es la opción más común y más sencilla de construir `DataFrames` en Spark. Para ello utilizaremos el objeto `DataFrame`Reader al que podemos acceder con desde nuestra sesión Spark mediante `spark.read`. Una vez tenemos este objeto, cargar datos a partir de un fichero CSV o JSON únicamente requiere invocar al método `csv` o `json`.

Veamos cómo cargar los datos de los pasajeros del Titanic a partir del fichero CSV. En el capítulo anterior, para poder crear un RDD de parejas necesitábamos encadenar varias transformaciones para dividir cada línea por comas y crear posteriormente un diccionario con los nombres de campos adecuados a partir de las listas de valores. Sin embargo, al usar `DataFrames` Spark se encargará de analizar cada línea, dividirla por comas y convertir cada parte a su tipo de datos de manera automática:

```
>>> df = spark.read.csv('data/Cap8/titanic.csv', header=True,
                        inferSchema=True)
>>> df.printSchema()
root
 |-- PassengerId: integer (nullable = true)
 |-- Survived: integer (nullable = true)
 |-- Pclass: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Age: double (nullable = true)
 |-- SibSp: integer (nullable = true)
 |-- Parch: integer (nullable = true)
 |-- Ticket: string (nullable = true)
 |-- Fare: double (nullable = true)
 |-- Cabin: string (nullable = true)
 |-- Embarked: string (nullable = true)
```

Como se ve, únicamente hemos necesitado pasar la ruta del fichero a leer y dos parámetros. El primero (`header=True`) indica que queremos que utilice la primera línea para conocer los nombres de columnas, como es usual en el formato CSV. El segundo parámetro (`inferSchema=True`) indica que queremos que Spark infiera qué tipo de datos se almacena en cada columna. Si no activamos este parámetro el resultado será muy similar, pero Spark dejará los valores de cada columna como cadenas de texto ya que evitará inspeccionarlos. El método `csv` admite una gran cantidad de parámetros adicionales para establecer el carácter separador (`sep`, por defecto una coma), la codificación (`encoding`, por defecto UTF-8), el carácter para

introducir citas que pueden contener al carácter separador (`quote`, por defecto las comillas dobles) o forzar a que las entradas cumplan un determinado esquema de datos definido vimos en la sección anterior. Por ello recomendamos al lector que consulte la documentación de la clase `DataFrameReader` para configurar adecuadamente la lectura del fichero CSV.

Además de ficheros CSV, el objeto `DataFrame` también permite leer con gran facilidad los ficheros JSON. En este caso únicamente debemos indicar la ruta del fichero y Spark hace el resto, analizando cada entrada e infiriendo el esquema de datos automáticamente. Por ejemplo, podemos cargar un fichero con información (muy simplificada) sobre distintos tweets con el siguiente código:

```
>>> df = spark.read.json('data/Cap8/tweets.json')
>>> df.printSchema()
root
 |-- RT_count: long (nullable = true)
 |-- text: string (nullable = true)
 |-- user: struct (nullable = true)
 |   |-- followers_count: long (nullable = true)
 |   |-- name: string (nullable = true)
 |   |-- verified: boolean (nullable = true)
>>> df.show()
+--------+-------------+---------------+
|RT_count|         text|           user|
+--------+-------------+---------------+
|       2| #Tengosueño  | [3, Pepe, true]|
|      45| #VivaElLunes |[15, Ana, false]|
|     100|¡Gol de Señor!|  [2, Eva, true]|
+--------+-------------+---------------+
```

El fichero `data/Cap8/tweets.json`, que se puede encontrar en el repositorio de código del libro, contiene únicamente 3 entradas, cada una en una línea. Tal y como se puede descubrir gracias a la inferencia del esquema de datos, contiene 3 campos, donde *user* contiene a su vez 3 campos anidados. Para cada campo ha inferido su tipo: número entero `double` para *RT_count* y *followers_count*, `str` para *text* y *name* y `bool` para el campo *verified*. Al igual que la lectura de ficheros CSV, al leer un fichero JSON se nos permitirá proporcionar un esquema para verificar que todas las entradas lo cumplen (parámetro `schema`), configurar si los nombres de campos deben estar entrecomillados (parámetro `allowUnquotedFieldNames`), permitir comillas simples (parámetro `allowSingleQuotes`), etc. Debido al alto número de parámetros recomendamos consultar la documentación de la clase `DataFrameReader` para configurar de manera precisa la lectura del fichero JSON.

Aparte de ficheros CSV y JSON, Spark también permite crear DataFrames de manera sencilla a partir de otros formatos de fichero del ecosistema Hadoop como OCR (método ocr) o Parquet (método parquet), o crear un DataFrame a partir de una tabla almacenada en una base de datos relacional accesible mediante JDBC (método jdbc). También permite cargar uno o más archivos de texto como un DataFrame de una única columna *value* (método text), ya sea creando una fila por línea de fichero o una fila por cada fichero. Se puede encontrar más información sobre estos métodos en la documentación de la clase DataFrameReader.

Almacenamiento de DataFrames

El almacenamiento de DataFrames es muy similar a su lectura, puesto que se centraliza a través de un objeto escritor de tipo DataFrameWriter. A la hora de almacenar un DataFrame df accederemos a su objeto escritor usando df.write. Este objeto escritor proporciona distintos métodos, dependiendo del formato de salida que deseemos.

Para almacenar el contenido de un DataFrame usando el formato CSV usaremos el método csv del objeto escritor. El único parámetro obligatorio que debemos pasar es la ruta donde almacenar la información, que debe ser en el sistema local o en un sistema distribuido como HDFS. Al igual que ocurría al almacenar RDDs, la ruta que pasamos no será un fichero sino el nombre de una carpeta. Esto es así porque todo DataFrame estará almacenado internamente como un RDD de objetos de tipo Row. Este RDD estará distribuido a lo largo del clúster, por lo que constará de varias particiones. Al almacenar un DataFrame, cada partición guardará su contenido en un fichero part-XXXXX-*.csv dentro de la carpeta elegida, donde XXXXX será el número de la partición. Por ejemplo, para crear un DataFrame de 2 columnas y 50 filas y almacenarlo en la carpeta /tmp/csv en formato CSV (usando los parámetros por defecto) ejecutaríamos el siguiente código:

```
>>> l = [('a',3.14)] * 50
>>> df = spark.createDataFrame(l, ['id','value'])
>>> df.write.csv('/tmp/csv')
```

Por defecto, cada línea estará separada por comas y los ficheros no incluirán la cabecera. Si quisiéramos cambiar el símbolo separador de valores utilizaríamos el parámetro sep, y para incluir las cabeceras en cada fichero utilizaríamos el parámetro header=True. Otro parámetro interesante es el modo de escritura mode, que nos permitirá configurar qué hacer si la carpeta ya existe: mediante append añadiremos los datos del DataFrame a los ficheros existentes, con overwrite los sobrescribiremos completamente, usando ignore evitaremos cualquier escritura si la

carpeta existe, y con `error` lanzaremos una excepción si la carpeta existía anteriormente.

De manera similar, para almacenar un `DataFrame` usando el formato JSON utilizaremos el método `json` de su objeto escritor. Este método acepta la ruta de la carpeta donde almacenar los datos, y al igual que `csv`, genera un fichero por cada partición del RDD subyacente. También nos permite elegir el modo de escritura a través del parámetro `mode`, que toma los mismos valores descritos anteriormente. Por ejemplo, para crear un `DataFrame` de 2 columnas y 50 filas y almacenarlo en formato JSON usando los parámetros por defecto ejecutaríamos el siguiente código:

```
>>> l = [('a',3.14)] * 50
>>> df = spark.createDataFrame(l, ['id','value'])
>>> df.write.json('/tmp/json')
```

Es importante darse cuenta de que el hecho de que un `DataFrame` se almacene en distintos ficheros no será un problema para recuperarlos posteriormente. Como ya vimos, los métodos de `spark.read` permiten pasar como ruta una carpeta del sistema de ficheros o incluso una expresión con comodines (`*`). Si quisiéramos volver a cargar el `DataFrame` a partir del directorio /tmp/json únicamente tendríamos que ejecutar:

```
>>> df = spark.read.json('/tmp/json')
>>> df.printSchema()
root
 |-- id: string (nullable = true)
 |-- value: double (nullable = true)
>>> df.count()
50
```

Como se puede ver, se ha reconstruido el esquema de datos original con dos columnas: *id* que contiene cadenas de texto y *value* que contiene un número decimal. Mediante el método `count` obtenemos el número de elementos del `DataFrame` que es 50 como en el `DataFrame` original que almacenamos. Para recuperar un `DataFrame` almacenado en formato CSV el procedimiento sería similar invocando a `spark.read.csv`, pero deberíamos utilizar los parámetros `header=True` junto con `inferSchema=True` para recuperar el esquema de datos original.

Además de los formatos CSV y JSON, el objeto escritor admite otros formatos del ecosistema Hadoop como OCR o Parquet, permite almacenar en tablas de bases de datos relacionales accesibles mediante JDBC, o almacenar las entradas en ficheros de texto. Este último caso únicamente será posible si el `DataFrame` está formado por una única columna que almacena una cadena de texto. Para ver más detalles sobre

estos métodos y sobre otros parámetros no tratados en este libro (como por ejemplo la posibilidad de comprimir los ficheros generados) recomendamos consultar la documentación del objeto `DataFrameWriter`.

Como comentario final, mencionar que también es posible exportar un `DataFrame` de Spark a uno de `pandas` utilizando el método `toPandas`. Este método recoge todos los datos del `DataFrame` y crea un `DataFrame` de `pandas` con el mismo esquema de datos. Sin embargo, es necesario tener en cuenta que eso implica que todos los datos del `DataFrame` se recopilarán en el proceso *driver*, por lo que al igual que el método `collect` de RDDs únicamente se podrá aplicar a `DataFrames` pequeños.

DataFrames y MongoDB

Además de cargar y salvar `DataFrames` utilizando los formatos de archivo usuales, Spark proporciona una integración muy sencilla con MongoDB que nos permite cargar `DataFrames` a partir de colecciones y volcar `DataFrames` a colecciones. Para poder utilizar esta integración primero debemos cargar el conector *MongoDB para Spark* a la hora de invocar a `pyspark`:

```
$ pyspark
    --packages org.mongodb.spark:mongo-spark-connector_2.11:2.2.2
```

El valor pasado al parámetro `packages` son las coordenadas Maven del conector que queremos cargar. No es necesario descargar nada en nuestro ordenador, ya que `pyspark` lo descargará automáticamente a partir de las coordenadas. En este caso cargaremos la versión `2.11:2.2.2`. El primer componente indica que queremos el conector que funciona con Scala 2.11, mientras que `2.2.2` es la versión concreta del conector. Según la documentación de MongoDB, esta versión es la adecuada para Spark versión 2.2.x y 2.3.x y para MongoDB versiones 2.6 o posteriores.

Para volcar un `DataFrame` a una colección MongoDB utilizaremos el objeto `DataFrameWriter`, de manera similar a cuando escribíamos ficheros. La principal diferencia es que ahora deberemos configurar de manera precisa el formato y las diversas opciones para realizar el volcado. Para mostrarlo con un ejemplo, crearemos un `DataFrame` con la población de 8 ciudades:

```
>>> ciudades = spark.createDataFrame([
      ("Madrid",3182981), ("Barcelona",1620809), ("Valencia",787808),
      ("Sevilla", 689434), ("Zaragoza", 664938), ("Málaga",569002),
      ("Murcia",443243), ("Palma",406492)], ["nombre","habitantes"])
```



```
>>> ciudades.show()
+---------+----------+
|   nombre|habitantes|
+---------+----------+
|   Madrid|   3182981|
|Barcelona|   1620809|
| Valencia|    787808|
|  Sevilla|    689434|
| Zaragoza|    664938|
|   Málaga|    569002|
|   Murcia|    443243|
|    Palma|    406492|
+---------+----------+
>>> ciudades.printSchema()
root
 |-- nombre: string (nullable = true)
 |-- habitantes: long (nullable = true)
```

Como se puede ver, se ha creado un `DataFrame` de 8 elementos y 2 columnas: *nombre* de tipo `str` y *habitantes* de tipo `long`. Para volcar este `DataFrame` a la colección *ciudades* de la base de datos *test* obtendremos el objeto `DataFrameWriter` y estableceremos el formato `"com.mongodb.spark.sql.DefaultSource"`, para utilizar MongoDB como fuente de datos. También deberemos configurar la dirección concreta de la colección MongoDB donde queremos volcar el `DataFrame` utilizando el URI completo: `mongodb://<host>:<puerto>/<base_de_datos>.<colección>`.

Concretamente, para volcar el `DataFrame ciudades` a la colección `test.ciudades` del servidor local ejecutaremos la siguiente instrucción:

```
>>> (ciudades.write.format("com.mongodb.spark.sql.DefaultSource")
                   .option("uri","mongodb://127.0.0.1/test.ciudades")
                   .save()
    )
```

Es importante incluir los paréntesis para indicar a Python de que, aunque está repartido en varias líneas, se trata de la misma instrucción. Como se puede observar, primero se configuran los parámetros concretos y finalmente se invoca a `save`. Para que esta instrucción tenga éxito la colección destino, en este caso `test.ciudades`, no debe existir en el servidor. Si lo que queremos es añadir a una colección ya existente, deberemos configurarlo expresamente con `mode("append")`:

```
>>> (ciudades.write.format("com.mongodb.spark.sql.DefaultSource")
                   .option("uri","mongodb://127.0.0.1/test.ciudades")
                   .mode("append").save()
    )
```

Cargar un `DataFrame` desde una colección MongoDB es igual de sencillo: se obtiene el objeto `DataFrameReader` desde el objeto `spark`, se configuran las opciones y finalmente se invoca a `read`:

```
>>> df = (spark.read.format("com.mongodb.spark.sql.DefaultSource")
              .option("uri","mongodb://127.0.0.1/test.ciudades")
              .load()
         )
>>> df.show()
+--------------------+----------+---------+
|                 _id|habitantes|   nombre|
+--------------------+----------+---------+
|[5af9a49a8773ad00...|   3182981|   Madrid|
|[5af9a49a8773ad00...|   1620809|Barcelona|
|[5af9a49a8773ad00...|    787808| Valencia|
|[5af9a49a8773ad00...|    689434|  Sevilla|
|[5af9a49a8773ad00...|    664938| Zaragoza|
|[5af9a49a8773ad00...|    569002|   Málaga|
|[5af9a49a8773ad00...|    443243|   Murcia|
|[5af9a49a8773ad00...|    406492|    Palma|
+--------------------+----------+---------+
>>> df.printSchema()
root
 |-- _id: struct (nullable = true)
 |    |-- oid: string (nullable = true)
 |-- altura: integer (nullable = true)
 |-- edad: integer (nullable = true)
 |-- nombre: string (nullable = true)
```

Los datos cargados son los mismos que habíamos volcado antes, con la excepción del atributo *_id* que ha sido añadido automáticamente al volcar el `DataFrame ciudades` a la colección MongoDB y que ahora ha sido recuperado junto con el resto de datos.

Finalmente, el conector MongoDB para Spark también nos permite aplicar una tubería para cargar un `DataFrame` a partir de una serie de operaciones sobre una colección. La manera de realizar esta carga es igual a la de recuperar una colección entera pero pasando una opción "`pipeline`" con la descripción de la tubería que queremos aplicar. Por ejemplo, si quisiéramos crear un `DataFrame` a partir de la colección `test.ciudades` pero únicamente cargar los documentos relativos a ciudades de más de 500000 habitantes ejecutaríamos el siguiente código:

```
>>> pipeline = "{'$match': {'habitantes': {$gt:500000}}}"
>>> masde500mil = (
      spark.read.format("com.mongodb.spark.sql.DefaultSource")
                .option("uri","mongodb://127.0.0.1/test.ciudades")
                .option("pipeline", pipeline).load()
    )
>>> masde500mil.show()
+--------------------+----------+---------+
|                 _id|habitantes|   nombre|
+--------------------+----------+---------+
|[5af9a49a8773ad00...|   3182981|   Madrid|
|[5af9a49a8773ad00...|   1620809|Barcelona|
|[5af9a49a8773ad00...|    787808| Valencia|
|[5af9a49a8773ad00...|    689434|  Sevilla|
|[5af9a49a8773ad00...|    664938| Zaragoza|
|[5af9a49a8773ad00...|    569002|   Málaga|
+--------------------+----------+---------+
```

Operaciones sobre DataFrames

Los `DataFrames` admiten una gran cantidad de operaciones que nos facilitarán el manejo de sus datos. En este apartado únicamente nos centraremos en las más interesantes (desde nuestro punto de vista), pero recomendamos consultar la documentación de la clase `DataFrame` para tener una visión completa de todos los métodos disponibles junto con los parámetros que admiten. A continuación, veremos con detalle operaciones para inspeccionar `DataFrames` y conocer qué datos almacenan, para seleccionar fragmentos de un `DataFrame` que nos interesen, para combinar la información de dos `DataFrames`, para calcular columnas nuevas a partir de los datos de un `DataFrame` y para poder realizar consultas SQL.

INSPECCIÓN DE DATAFRAMES

A lo largo de los ejemplos de esta sección hemos utilizado de manera intensiva el método `printSchema` que nos permite conocer el esquema de datos de un `DataFrame`. Este método no devuelve nada, sino que imprime por pantalla el esquema de datos del `DataFrame` con todo detalle.

Otro método interesante es `count`, que como hemos visto anteriormente nos devuelve el número de filas que tiene un `DataFrame`. Esta información es muy importante a la hora de estimar el tamaño de un `DataFrame`, por ejemplo, para decidir si es "sensato" recuperarlo como `DataFrame` de la biblioteca `pandas` y trabajar con él en el proceso *driver*. El siguiente ejemplo crea un `DataFrame` de 3 filas y muestra su tamaño:

```
>>> l = [('a',3.14), ('b',2.0), ('c',4.5)]
>>> df = spark.createDataFrame(l, ['id','value'])
>>> df.count()
3
```

Otro método muy útil para conocer los datos que almacena un `DataFrame` es `describe`. Este método acepta como parámetro una lista de nombres de columnas y genera un nuevo `DataFrame` resumen, que contendrá información de cada columna:

1. El número de filas que tienen un valor no vacío (*count*).
2. El valor promedio (*mean*).
3. La desviación típica (*stddev*).
4. El valor mínimo (*min*).
5. El valor máximo (*max*).

Si una columna almacena valores no numéricos, su valor promedio y desviación típica se establecen a `null`, ya que en esos casos no estarían definidas. Aunque pueda parecer extraño, para columnas no numéricas sí que se calcula su máximo y mínimo utilizando la noción de orden que utilice Python por defecto. Por ejemplo, en el caso de listas, cadenas de texto o tuplas se utilizará el orden lexicográfico.

Gracias al método `describe` podemos inspeccionar de manera sencilla el conjunto de datos sobre los pasajeros del Titanic:

```
>>> df = spark.read.csv('data/Cap8/titanic.csv', header=True,
                        inferSchema=True)
>>> df.describe(['Age','Fare','Sex','Cabin']).show()
+-------+--------------+-------------+------+-----+
|summary|           Age|         Fare|   Sex|Cabin|
+-------+--------------+-------------+------+-----+
|  count|           714|          891|   891|  204|
|   mean| 29.6991176470| 32.204207968|  null| null|
| stddev|14.52649733233|49.6934285971|  null| null|
|    min|          0.42|          0.0|female|  A10|
|    max|          80.0|     512.3292|  male|    T|
+-------+--------------+-------------+------+-----+
```

En este caso hemos calculado el resumen de las columnas *Age*, *Fare*, *Sex* y *Cabin*. Lo que devuelve el método `describe` es un nuevo `DataFrame`, así que para visualizarlo debemos invocar a su método `show`. Observando la fila *count* de este nuevo `DataFrame` descubrimos que *Fare* y *Sex* no tienen valores vacíos, *Age* tiene 177 valores vacíos (891-712) y la columna *Cabin* está prácticamente vacía. También

nos permite conocer de un vistazo que las edades están en el rango [0.42, 80.0] con media de 19.7 y desviación típica de 14.5. Esta información es muy interesante ya que nos permitiría escalar los valores de las columnas *Age* y *Fare* (aunque como veremos más adelante, es más sencillo hacerlo usando SparkML), además de descartar la columna *Cabin* porque tiene demasiados valores nulos.

FILTRADO DE DATAFRAMES

En este apartado veremos ciertas operaciones sobre `DataFrames` que nos permitirán eliminar algunas partes y quedarnos únicamente con la información que nos interesa. La operación más sencilla de esta familia es el método `drop`, que nos permite eliminar una o varias columnas de un `DataFrame`. Esta operación no modifica el `DataFrame` original, que es inmutable, sino que crea uno nuevo con las columnas esperadas. En el siguiente ejemplo cargamos los datos de los pasajeros del Titanic y eliminamos las columnas *PassengerId*, *Name* y *Cabin*. Por concisión, para mostrar únicamente los nombres de las columnas de un `DataFrame` accedemos a su atributo `columns` en lugar de mostrar el esquema completo usando `printSchema`:

```
>>> titanic = spark.read.csv('data/titanic.csv', header=True,
                             inferSchema=True)
>>> titanic.columns
['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked']
>>> df = titanic.drop('PassengerId','Name','Cabin')
>>> df.columns
['Survived', 'Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket',
 'Fare', 'Embarked']
```

Como se puede ver, el `DataFrame titanic` original tiene 12 columnas, mientras que el `DataFrame df` resultante tiene 9, donde se han eliminado únicamente las columnas eliminadas *PassengerId*, *Name* y *Cabin*. Otra manera alternativa de reducir el número de columnas de un `DataFrame` es seleccionar únicamente aquellas columnas que nos interesan con el método `select`. Este método acepta una secuencia de nombres de columnas y genera un nuevo `DataFrame` que conserva únicamente las columnas elegidas (el método `select` también permite realizar operaciones avanzadas sobre columnas, como veremos más adelante en este capítulo). Por ejemplo, podríamos seleccionar únicamente las columnas *Survived*, *Pclass* y *Age* del conjunto de datos sobre supervivientes del Titanic de la siguiente forma:

```
>>> titanic = spark.read.csv('data/titanic.csv', header=True,
                             inferSchema=True)
>>> print(titanic.columns)
['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked']
>>> df2 = titanic.select('Survived','Pclass','Age')
>>> print(df2.columns)
['Survived', 'Pclass', 'Age']
```

Otra operación útil para filtrar `DataFrames` es `dropDuplicates`, que permite eliminar filas repetidas. Si no pasa ningún parámetro, `dropDuplicates` considerará todas las columnas a la hora de decidir cuándo dos filas son duplicadas. Sin embargo, se puede pasar una lista de nombres de columnas y únicamente se inspeccionarán esos valores a la hora de considerar filas duplicadas. Por ejemplo, podemos eliminar posibles duplicados en nuestro `DataFrame titanic` del ejemplo anterior y comprobar cuántas filas se han eliminado con el siguiente código:

```
>>> titanic.count()
891
>>> df = titanic.dropDuplicates()
>>> df.count()
891
```

En este caso vemos que `titanic` tiene 891 filas, y tras eliminar duplicados considerando todas las columnas seguimos teniendo 891 filas en `df`. Esto quiere decir que no existen dos filas con exactamente los mismos valores en todas sus columnas. Si hubiésemos sido más restrictivos y contásemos únicamente la columna *Sex* para considerar duplicados, el resultado sería más drástico:

```
>>> df = titanic.dropDuplicates(['Sex'])
>>> df.count()
2
```

En este caso el resultado es 2 porque únicamente hay dos valores en la columna *Sex*: `'female'` y `'male'`. En este caso, dropDuplicates considera todas las filas con valor `'female'` o `'male'` como duplicados, por lo que únicamente deja una de cada valor.

Otra operación interesante a la hora de limpiar nuestros `DataFrames` es `dropna`, que eliminará aquellas filas que contengan valores vacíos de manera similar a la operación homónima en `Pandas`. Este método es bastante versátil y permite configurar de manera muy detallada la manera en la que se decide si una fila debe o no ser descartada. Por un lado, dispone de un parámetro `how` que indica si se deben

eliminar las filas que tengan todos sus valores nulos (valor `'all'`) o si únicamente es necesario que contenga algún valor vacío para proceder a su eliminación (valor `'any'`). Por otro lado, también permite definir cuál es el mínimo número de valores no vacíos que debe tener una fila para conservarla en el `DataFrame` (parámetro `thresh`). Este parámetro invalidaría el valor pasado para `how`. Por último, permite recibir una lista de nombres de columnas sobre las que buscar valores vacíos en su parámetro `subset`. Por defecto, la operación `dropna` elimina aquellas filas que tengan un valor vacío en alguna de sus columnas (es decir, `how='any'`).

Podemos limpiar nuestro `DataFrame` titanic y quedarnos únicamente con aquellas filas que contienen datos en todas las columnas. Para ello ejecutaríamos:

```
>>> df = titanic.dropna()
>>> df.count()
183
>>> df = titanic.drop('Cabin').dropna()
>>> df.count()
712
```

Ejecutando `dropna` directamente sobre el `DataFrame titanic` obtenemos únicamente 183 filas. Esto es así porque, como vimos al tratar el método `describe`, la columna *Cabin* tiene únicamente 204 valores no vacíos. Combinar esta columna con otras a la hora de buscar filas con algún valor vacío hace que este número aumente aún más. Si excluimos esta columna antes de filtrar las filas nulas obtendremos 712 resultados, que coincide con el valor obtenido al realizar este proceso sobre RDDs.

Por último, la operación más potente a la hora de filtrar `DataFrames` es `filter`. Esta operación recibe como parámetro una condición y filtra el `DataFrame` conservando únicamente aquellas filas que la cumplen. Por ejemplo, podemos obtener todos los pasajeros del Titanic que sobrevivieron con la siguiente instrucción:

```
>>> df = titanic.filter('Survived = 1')
>>> df.count()
342
```

En este caso hemos expresado la condición como una cadena de texto que usa la misma sintaxis que SQL. Normalmente es la opción más cómoda, ya que es sencilla de entender y además la inmensa mayoría de la gente está familiarizada con esta sintaxis. Sin embargo, también podemos expresar mediante una condición booleana involucrando objetos de tipo `pyspark.sql.Column`. Para obtener estos objetos a partir de un `DataFrame` podemos usar el punto, como si se tratase de un atributo (por ejemplo, `titanic.Survived`) o usando los corchetes para indexar usando una

cadena de texto (`titanic['Survived']`). Usando este tipo de notación la consulta anterior se expresaría así:

```
>>> df = titanic.filter(df.Survived == 1)
```

La condición utilizada para filtrar se puede componer e involucrar distintas comprobaciones sobre distintas columnas. Por ejemplo, podríamos estar interesados únicamente en las supervivientes de más de 20 años:

```
>>> df = titanic.filter( 'Survived = 1 AND Sex = "female" AND
                          Age > 20')
>>> df.count()
144
>>> df = titanic.filter( (df.Survived == 1) &
                         (df['Sex'] == 'female') &
                         (df.Age > 20))
>>> df.count()
144
```

A la hora de representar la condición como una expresión SQL hemos utilizado el operador `AND`, mientras que al utilizar columnas hemos utilizado el operador `&`. De la misma manera el operador `OR` en SQL se traduciría en el operador barra vertical (`|`) para columnas, y el operador `NOT` en el operador tilde (`~`). Es importante darse cuenta de que en la expresión SQL hemos necesitado utilizar comillas dobles para expresar la cadena `"female"`, puesto que toda la expresión estaba encerrada entre comillas simples.

La elección entre condiciones en notación SQL o usando operaciones sobre columnas dependerá del conocimiento que tenga el usuario de cada una de las opciones, puesto que ambas alternativas tienen la misma expresividad. No obstante, recomendamos al lector revisar los distintos operadores disponibles en la documentación de la clase `Column` para descubrir si alguna agiliza la escritura de la condición de su consulta. En caso de duda, nuestro consejo es utilizar la notación SQL debido a su mayor difusión y claridad.

COMBINACIÓN DE DATAFRAMES

En algunas ocasiones tendremos la información dividida en distintos `DataFrames` y querremos combinarla para crear un único `DataFrame`. Para ello SparkSQL nos proporciona distintos métodos dependiendo lo que necesitemos: las operaciones binarias usuales de conjuntos (unión, intersección y diferencia) y la reunión (*join*).

Para poder combinar dos DataFrames utilizando operaciones de conjuntos es imprescindible que tengan esquemas de datos compatibles, es decir, que contengan el mismo número de columnas y que los tipos de las columnas en la misma posición en ambos DataFrames sean compatibles. A la hora de decidir si dos columnas son compatibles SparkSQL trata de ser lo más flexible posible, por ejemplo, considera que los tipos long y double son compatibles, expresando todos los valores en el tipo más amplio que es double. De la misma manera, y aunque pueda resultar inesperado, considera compatibles los tipos long y string, ya que los números se pueden expresar como cadenas de texto con su representación en decimal. El nombre que tenga cada columna no afecta a la compatibilidad de esquemas de datos, y en caso de haber alguna colisión SparkSQL utilizará por defecto los nombres del primer DataFrame.

Una vez que tenemos dos DataFrames con esquemas compatibles podemos unirlos con el método union, que acepta como parámetro el segundo DataFrame:

```
>>> df1 = spark.createDataFrame([(1,'ana'), (2,'jose')],
            ['id','nombre'])
>>> df2 = spark.createDataFrame([(3,'marta'),(1,'ana')],
            ['id','nombre'])
>>> df = df1.union(df2)
>>> df.show()
+---+------+
| id|nombre|
+---+------+
|  1|   ana|
|  2|  jose|
|  3| marta|
|  1|   ana|
+---+------+
```

En este caso hemos creado dos DataFrames con dos filas y dos columnas: *id* de tipo entero y *nombre* de tipo cadena de texto. Como ambos esquemas de datos son compatibles, la unión tiene éxito, generando un DataFrame de 4 filas y las mencionadas columnas. Como se puede ver, la fila (1,'ana') está repetida porque aparecía tanto en el DataFrame df1 como df2 y el método union no elimina duplicados. Si quisiéramos quedarnos con los elementos únicos, deberíamos aplicar el método dropDuplicates que hemos visto anteriormente al resultado de la unión.

De la misma manera se pueden aplicar intersección y diferencia a DataFrames con esquemas de datos compatibles. Por ejemplo, podríamos obtener la intersección de los anteriores DataFrames df1 y df2, lo que conservaría únicamente aquellos elementos que aparecen en ambos DataFrames:

```
>>> df = df1.intersect(df2)
>>> df.show()
+---+------+
| id|nombre|
+---+------+
|  1|   ana|
+---+------+
```

En este caso el único elemento que aparece en ambos `DataFrames` es `(1,'ana')`. Para calcular la intersección, SparkSQL comprueba que los valores coincidan en todas las columnas, por lo que el elemento `(1,'ana')` sería distinto de `(1,'ANA')` y de `(10,'ana')`. En el repositorio de código que acompaña al libro se pueden ver ejemplos de diferencia de `DataFrames` (método `subtract`) y varias situaciones de compatibilidad e incompatibilidad de esquemas de datos a la hora de combinar `DataFrames` con operaciones de conjuntos.

SparkSQL también permite combinar los datos de dos `DataFrames` fusionando filas que tienen los mismos valores en ciertas columnas. Esta operación de reunión, con la misma funcionalidad que el operador JOIN de SQL, se invoca a través del método `join`. Por defecto este método realiza *inner join* de los dos `DataFrames` por ser el tipo más usual, y este es el tipo de combinación que utilizaremos en los ejemplos que siguen, pero este método admite un parámetro `how` que permite realizar cualquier otro tipo de combinación: `'outer'`, `'left'`, `'right'`, etc. Para más información sobre los tipos de combinación disponibles recomendamos revisar la documentación de la clase `DataFrame`.

El siguiente ejemplo muestra una combinación *inner* usando la columna *id* de dos `DataFrames`, uno que almacena el nombre de los usuarios y otro que almacena su edad:

```
>>> users = spark.createDataFrame([(1,'ana'),(2,'jose')],
              ['id','nombre'])
>>> age = spark.createDataFrame([(1,36),(2,30)],['id','edad'])
>>> df = users.join(age,'id')
>>> df.show()
+---+------+----+
| id|nombre|edad|
+---+------+----+
|  1|   ana|  36|
|  2|  jose|  30|
+---+------+----+
```

Como ambos `DataFrames` contienen filas con identificadores 1 y 2 dichas filas se fusionan en una sola con las columnas de cada `DataFrame`. Como la columna *id* que

dirige la combinación aparece en los dos DataFrames no se duplicará en el resultado, sino que se conservará únicamente una de ellas. En este ejemplo las dos columnas *id* contenían valores de tipo entero, pero el método join seguirá funcionando aunque estas columnas contengan tipos diferentes siempre que sean compatibles. Este es el caso de los números enteros y cadenas de texto que mencionamos anteriormente: si el DataFrame users almacena los identificadores como cadenas de texto y *age* como enteros, la combinación seguirá generando los mismos resultados, pero usando el tipo más general string para los identificadores del DataFrame resultante.

```
>>> users = spark.createDataFrame([('1','ana'), ('2','jose')],
              ['id','nombre'])
>>> age = spark.createDataFrame([(1,36),(2,30)],['id','edad'])
>>> df = users.join(age ,'id')
>>> df.printSchema()
root
 |-- id: string (nullable = true)
 |-- nombre: string (nullable = true)
 |-- edad: long (nullable = true)
>>> df.show()
+---+------+----+
| id|nombre|edad|
+---+------+----+
|  1|   ana|  36|
|  2|  jose|  30|
+---+------+----+
```

En los ejemplos que hemos visto hasta ahora la combinación se refiere únicamente a una columna que existe en ambos DataFrames. Sin embargo, en algunas ocasiones querremos combinar filas cuyos valores coincidan en 2 o más columnas. En esos casos deberemos pasar una lista de nombres de columnas en lugar de un único nombre de columna:

```
>>> users = spark.createDataFrame([(1,'ana','golf'),
      (2,'jose','polo',)],['id','nombre','deporte'])
>>> age = spark.createDataFrame([(1,'eva',33),
      (2,'jose',30)],['id','nombre','edad'])
>>> df = users.join(age,["id", "nombre"])
>>> df.show()
+---+------+-------+----+
| id|nombre|deporte|edad|
+---+------+-------+----+
|  2|  jose|   polo|  30|
+---+------+-------+----+
```

En este caso estamos haciendo la combinación *inner* en las columnas *id* y *nombre*. En ambos `DataFrames` hay una fila con *id* igual a 1, pero esas filas difieren en el valor de su columna *nombre* (`users` contiene `'ana'` mientras que `age` contiene `'eva'`). Como esas dos filas no contienen los mismos valores en las dos columnas a la vez no aparecen en el resultado de la combinación, que únicamente contiene la fila relativa al usuario de identificador 2 y nombre `'jose'`.

Por último, mencionar que el método `join` también admite combinación de `DataFrames` sobre columnas que tienen diferentes nombres. En estos casos en lugar de usar nombres de columnas utilizaremos expresiones booleanas sobre columnas tal y como veíamos en el método `filter`. Por ejemplo, para hacer la combinación entre un `DataFrame df1` con columna *id* y otro `DataFrame df2` con columna *ident* tendríamos que utilizar la expresión `df1.id == df2.ident`. Este tipo de expresiones se puede ampliar a varias columnas mediante el operador de conjunción `&`, por ejemplo `df1.id == df2.ident & df1.age == df2.edad`. La potencia de este tipo de expresiones va más allá, ya que permiten utilizar cualquier operador como de comparación (>, !=, <=, etc.) y cualquier operador booleano. Por ejemplo, imaginemos que tenemos un `DataFrame age` que, debido a que es el resultado de fusionar varias fuentes de datos, contiene dos columnas con identificadores: *id1* e *id2*. En algunos casos ambos valores coinciden, en otros casos alguno es vacío, y en otros casos difieren. En caso de que el valor de *id1* sea no vacío querríamos considerarlo como válido e ignorar *id2*, y en caso de que este valor fuera vacío consideraríamos el valor de *id2*. Podríamos realizar una combinación *inner* utilizando esta condición compleja representándola como una expresión booleana `cond` que utiliza los operadores `|` y `&`:

```
>>> users = spark.createDataFrame([(1,'ana'),(2,'jose')],
              ['id','nombre'])
>>> age = spark.createDataFrame([(None,1,33),(2,5,30)],
            ['id1','id2','edad'])
>>> cond = ( (age.id1.isNotNull() & (users.id==age.id1)) |
             (age.id1.isNull() & (users.id==age.id2))
           )
df = users.join(age, cond)
df.show()
+---+------+----+---+----+
| id|nombre| id1|id2|edad|
+---+------+----+---+----+
|  1|   ana|null|  1|  33|
|  2|  jose|   2|  5|  30|
+---+------+----+---+----+
```

TRANSFORMACIÓN DE DATAFRAMES

Además de mezclar `DataFrames` y seleccionar aquellas columnas que nos interesen, SparkSQL también permite realizar operaciones entre distintas columnas para obtener nuevas columnas que agregan esos valores. Este tipo de operaciones se realiza con el método `selectExpr`, que acepta cadenas de texto definiendo cada columna en el nuevo `DataFrame`. Para expresar las operaciones a realizar en cada columna se utiliza una sintaxis similar a las consultas SQL clásicas, lo que las hace particularmente cómodas para usuarios con experiencia en bases de datos relacionales. En este apartado usaremos para los ejemplos el `DataFrame titanic` que contiene los pasajeros del Titanic, eliminando algunas entradas con valores vacíos. Concretamente supondremos que el `DataFrame` se ha creado de la siguiente manera:

```
>>> titanic = spark.read.csv('data/titanic.csv', header=True,
      inferSchema=True).drop('Cabin').dropna()
```

Una de las posibles transformaciones que podemos realizar es procesar `titanic` para quedarnos únicamente con unas pocas columnas: la columna *Survived*, una columna *Family* que agrupa todos los familiares (suma de las columnas *SibSp* y *Parch*) y la edad representada en meses en lugar de años. Toda esta transformación se realizaría en una única invocación:

```
>>> titanic.selectExpr("Survived","SibSp + Parch AS Family",
      "Age * 12 AS Age").show(3)
+--------+------+-----+
|Survived|Family|  Age|
+--------+------+-----+
|       0|     1|264.0|
|       1|     1|456.0|
|       1|     0|312.0|
+--------+------+-----+
```

Para incluir una columna en el resultado únicamente tenemos que escribir su nombre, y para las nuevas columnas tendremos que escribir la operación que queremos realizar usando el nombre de las columnas. Así, para sumar las columnas *SibSp* y *Parch* usaremos `'SibSp + Parch'`, y para obtener el número de meses a partir del número de años escribiremos `'Age * 12'`. Si no añadimos nada más, las columnas nuevas tendrán como nombre exactamente la operación realizada para obtenerlas, es decir, `'SibSp + Parch'` y `'Age * 12'`, respectivamente. Como esto es bastante incómodo, se suele dar un nombre adecuado a las nuevas columnas mediante el operador `AS`, como se puede ver en el ejemplo: la columna con el número de familiares se llama *Family* y la edad en meses *Age*. Obsérvese que en este

caso no hay colisión con la columna original *Age* que contenía la edad en años ya que no ha sido seleccionada para aparecer en el `DataFrame` resultado.

En ocasiones necesitaremos aplicar funciones que no se pueden expresar mediante operadores predeterminados. Para solventar estas necesidades SparkSQL permite utilizar *funciones definidas por el usuario* (UDFs según sus siglas en inglés *user defined function*). Estas funciones se definen en el propio lenguaje de programación que estemos usando (Python en este caso, pero sería similar en Scala, Java o R) y se deben registrar con un nombre concreto. A partir de ese momento podremos incluirlas sin problema en las expresiones pasadas al método `selectExpr`. Veamos tres ejemplos de UDFs con distinta aridad y complejidad que se podrían aplicar al conjunto de datos de supervivientes del Titanic.

Según hemos visto en el conjunto de datos, el sexo se representa en la columna *Sex* como cadenas de texto `'female'` y `'male'`. Aunque así es más fácil de entender por humanos, si en algún paso posterior queremos aplicar algún algoritmo de aprendizaje automático deberemos transformarlo a números naturales. En este caso querremos representar `'female'` como 0 y `'male'` como 1. Como veremos en la siguiente sección, SparkML proporciona una transformación para realizar esta operación de manera automática, sin embargo, veremos cómo realizar esta transformación de manera manual. Para ello lo primero que debemos hacer es definir una función `sex_to_num` en Python que recibe la cadena de texto y devuelve el número adecuado. Podemos ser precavidos y considerar que el valor pasado puede ser vacía (`None`), en cuyo caso devolveremos ese valor. Esta función se podría definir como sigue:

```
def sex_to_num(s):
    ret = None
    if s == 'female':
        ret = 0
    elif s == 'male':
        ret = 1
    return ret
```

Una vez que tenemos la función Python debemos registrarla con un nombre y un tipo de retorno para que SparkSQL pueda utilizarla y verificar que los valores devueltos al utilizarla son compatibles con los declarados. El registro de funciones se realiza mediante la función `spark.udf.register`, que recibe 3 parámetros: el nombre que se dará a la función, la función Python que queremos registrar y el tipo de datos de los resultados de dicha función. En esta ocasión queremos registrar la función Python `sex_to_num` con el mismo nombre, y el valor devuelto será un entero (`IntegerType`). Como se puede ver, los tipos de datos se representan usando los mismos objetos que vimos a la hora de definir esquemas de tipos.


```
>>> from pyspark.sql.types import IntegerType
>>> spark.udf.register("sex_to_num", sex_to_num, IntegerType())
```

A partir de este momento disponemos de una UDF llamada *sex_to_num* que podemos incluir en las invocaciones a `selectExpr`. Para ver el funcionamiento vamos a crear un nuevo `DataFrame` a partir de `titanic` que contiene la columna *Sex* original además de otra llamada *Sex_num* creada usando la UDF *sex_to_num* a la columna *Age*:

```
>>> titanic.selectExpr("Sex", "sex_to_num(Sex) AS Sex_num").show(3)
+------+-------+
|   Sex|Sex_num|
+------+-------+
|  male|      1|
|female|      0|
|female|      0|
+------+-------+
```

La UDF *sex_to_num* acepta un único valor, pero podemos definir UDFs que procesen dos o más valores. Por ejemplo, podríamos querer resumir las columnas *SibSp* y *Parch* en una única columna *Max_Family* que contenga el valor máximo de ambas. Como Python ya dispone de la función predefinida `max` para calcular el máximo de varios valores, no será definir ninguna función Python pero sí registrarla con un nombre concreto. En este caso vamos a calcular el máximo de dos valores enteros, así que la registraremos con el nombre *max_int* y con tipo retorno `IntegerType`. Para invocarla dentro de `selectExpr` simplemente pasaremos como parámetros los dos nombres de columnas:

```
>>> spark.udf.register("max_int", max, IntegerType())
>>> titanic.selectExpr("SibSp", "Parch",
      "max_int(SibSp, Parch) AS Max_Family").show(3)
+-----+-----+----------+
|SibSp|Parch|Max_Family|
+-----+-----+----------+
|    1|    0|         1|
|    1|    0|         1|
|    0|    0|         0|
+-----+-----+----------+
```

Por último, otra operación que nos podría resultar útil sería el escalado de los valores las columnas en el rango [0, 1]. Para ello necesitaremos conocer el valor mínimo (`minv`) y máximo (`maxv`) de la columna y aplicar la siguiente operación a cada valor:

```
def scale(n,minv,maxv):
    return (n - minv) / (maxv - minv)
```

Esta función Python acepta 3 argumentos, pero la UDF que queremos utilizar debe aceptar únicamente uno. Esto no es un problema, ya que podemos registrar una función anónima que es una versión especializada de `scale` para aplicar escalado en un rango fijado. Imaginemos que tenemos la edad mínima y máxima almacenadas en las variables `min_age` y `max_age`, respectivamente (estos valores se podrían obtener directamente del `DataFrame` de `pandas` generado por `describe`). Con estos valores podremos definir una UDF *scale_Age* a partir de una función anónima que acepta un único valor y lo escala en el rango `[min_age, max_age]`. Como en el `DataFrame titanic` la edad es un número real, estableceremos el valor retorno de esta función a `DoubleType`. Una vez definida la UDF *scale_Age*, utilizarla será igual que la UDF *sex_to_num* del ejemplo anterior:

```
>>> from pyspark.sql.types import DoubleType
>>> spark.udf.register("scale_Age",
      lambda x: scale(x, min_age, max_age), DoubleType())
>>> titanic.selectExpr("Age", "scale_Age(Age) AS Scaled_Age").show(3)
+----+------------------+
| Age|         Scaled_Age|
+----+------------------+
|22.0| 0.2711736617240513|
|38.0| 0.4722292033174164|
|26.0|0.32143754712239253|
+----+------------------+
```

En los ejemplos vistos hasta ahora hemos utilizado como parámetros de `selectExpr` expresiones codificadas como cadenas de texto que siguen la sintaxis SQL. Como ya hemos visto en otros casos, SparkSQL permite también el uso de expresiones de columnas en su lugar, y la transformación de columnas no es una excepción. La única diferencia es que en este caso debemos invocar a `select` en lugar de `selectExpr`. Este método ya fue presentado como solución para seleccionar qué columnas deben aparecer en el nuevo `DataFrame`, pero veremos que tiene una potencia similar a `selectExpr`. Por ejemplo, podemos seleccionar la columna *Survived*, crear una nueva columna *Family* sumando *SibSp* y *Parch* y transformar la edad a meses en lugar de años:

```
>>> titanic.select(titanic.Survived,
      (titanic.SibSp + titanic.Parch).alias("Family"),
      (titanic.Age * 12).alias("Age")).show(3)
+--------+------+-----+
|Survived|Family|  Age|
+--------+------+-----+
|       0|     1|264.0|
|       1|     1|456.0|
|       1|     0|312.0|
+--------+------+-----+
```

Es importante darse cuenta de que en este caso debemos definir el renombrado de columnas utilizando el método `alias` de la clase `Column`, que es equivalente al operador `AS` que usábamos con `selectExpr`. Las operaciones aritméticas como sumar dos columnas o multiplicar una columna por un escalar devuelven un objeto de tipo `Column`, así que podemos renombrarlo con el método `alias` sin problema.

Al igual que `selectExpr`, `select` también permite el uso de UDFs mezcladas con las operaciones entre columnas. En este caso en lugar de registrarlas con un nombre mediante la función `spark.udf.register`, lo que haremos es crear un objeto funcional a partir de la función Python usando la función `pyspark.sql.functions.udf`. Este objeto encapsula la función Python junto con el tipo de retorno, y es el que debemos utilizar en nuestras expresiones entre columnas. Como estos objetos funcionales no se registran con ningún nombre, la función `udf` solo acepta dos parámetros: la función a encapsular y el tipo de retorno. A modo de ejemplo, las UDFs usadas anteriormente se encapsularían como sigue, considerando que tenemos las variables `min_age` y `max_age` definidas:

```
>>> from pyspark.sql.functions import udf
>>> sex_to_num_UDF = udf(sex_to_num, IntegerType())
>>> max_int_UDF = udf(max, IntegerType())
>>> scale_Age_UDF = udf(lambda x : scale(x, min_age, max_age),
                        DoubleType())
```

Como se puede ver, la creación de estos objetos funcionales es muy similar al registro de UDFs, con la única diferencia de que evitamos asignarles un nombre. Eso sí, debemos almacenarlos en una variable (en este caso `sex_to_num_UDF`, `max_int_UDF` y `scale_Age_UDF`) para poder utilizar estos objetos funcionales en las expresiones entre columnas. Su uso se puede ver en el siguiente fragmento de código que crea un `DataFrame` con las columnas *Scaled_Age*, *Sex_Num* y *Max_Family* de manera similar a los ejemplos anteriores.

```
>>> titanic.select(
      scale_Age_UDF(titanic.Age). alias("Scaled_Age"),
      sex_to_num_UDF(titanic.Sex).alias("Sex_Num"),
      max_int_UDF(titanic.SibSp,titanic.Parch).alias("Max_Family")
    ).show(3)
+------------------+-------+----------+
|         Scaled_Age|Sex_Num|Max_Family|
+------------------+-------+----------+
| 0.2711736617240513|      1|         1|
| 0.4722292033174164|      0|         1|
|0.32143754712239253|      0|         0|
+------------------+-------+----------+
```

Por último, SparkSQL también permite agrupar todas las filas que tengan el mismo valor en una o varias columnas y agregar sus valores aplicando alguna función como la suma, el mínimo, la media, contar el número de filas seleccionadas, etc. Para realizar este tipo de operaciones aplicaremos el método `groupBy`, que nos devolverá un objeto de tipo `GroupedData` sobre el que podremos invocar los métodos deseados. El método `groupBy` acepta cero, uno o varios parámetros que son los nombres de columnas (u objetos de tipo `Column`) sobre los que hacer la agrupación. Si no se pasa ningún parámetro, se agruparán todas las filas del `DataFrame`. Si se pasa uno o más parámetros, se agruparán las filas que contengan los mismos valores en dichas columnas. Una vez obtenido el objeto de tipo `GroupedData` podemos aplicar los métodos `sum`, `min`, `avg`, `count`, etc., que deseemos. En este apartado veremos únicamente algunos, pero recomendamos consultar la documentación de la clase `GroupedData` para obtener un listado de todas las operaciones disponibles.

Por ejemplo, usando agregaciones podemos calcular el número de filas del `DataFrame titanic` y también el número de supervivientes totales. Para esto último aprovecharemos que la columna *Survived* contiene un 1 si el pasajero sobrevivió y 0 en caso contrario, por lo que únicamente tenemos que sumar los valores de dicha columna:

```
>>> titanic.groupBy().count().show()
+-----+
|count|
+-----+
|  712|
+-----+
>>> titanic.groupBy().sum('Survived').show()
+-------------+
|sum(Survived)|
+-------------+
|          288|
+-------------+
```

Como se puede ver, en el primer caso no indicamos ninguna columna para realizar la agregación porque queremos procesar todas las filas del `DataFrame`. Para calcular el número de filas invocamos directamente a `count`, mientras que a la hora de calcular el número de supervivientes sumamos con `sum` los valores de la columna *Survived*. Si quisiéramos obtener datos más precisos, podríamos calcular el número de supervivientes de cada clase. Para ello, agruparíamos por la columna *Pclass* y sumaríamos la columna *Survived* como antes:

```
>>> titanic.groupBy('Pclass').sum('Survived').show()
+------+-------------+
|Pclass|sum(Survived)|
+------+-------------+
|     1|          120|
|     3|           85|
|     2|           83|
+------+-------------+
```

En estos ejemplos hemos aplicado los métodos de agregación a una sola columna, pero es posible aplicar una misma función a varias columnas a la vez. Por ejemplo, podemos sumar el número de supervivientes y el precio total de los billetes pagados por cada clase del barco. Para ello únicamente debemos pasar varios nombres de columna a la función `sum`:

```
>>> titanic.groupBy('Pclass').
      sum('Survived', 'Fare').show()
+------+-------------+-----------------+
|Pclass|sum(Survived)|         sum(Fare)|
+------+-------------+-----------------+
|     1|          120| 16200.85429999999|
|     3|           85| 4696.449500000006|
|     2|           83|3714.5791999999997|
+------+-------------+-----------------+
```

En otras ocasiones querremos agrupar los datos de un `DataFrame` por ciertas columnas, pero aplicar funciones diferentes de agregación con esos datos. En esos casos deberemos utilizar el método `agg` sobre el objeto de tipo `GroupedData` devuelto por `groupBy`. El método `agg` acepta como parámetro un diccionario donde las claves son el nombre de columna y el valor asociado es la operación a realizar, ambos representados como cadenas de texto. Por ejemplo, podemos agrupar los datos por clase y calcular el número total de pasajeros y cuántos sobrevivieron. Para ello tendríamos que aplicar `count` sobre cualquier columna para calcular el total de pasajeros y `sum` sobre la columna *Survived* para obtener los supervivientes. Estas

operaciones se representarían como un diccionario de dos entradas tal y como se muestra a continuación:

```
>>> titanic.groupBy('Pclass').agg(
      {'*':'count', 'Survived':'sum'}).show()
+------+-------------+--------+
|Pclass|sum(Survived)|count(1)|
+------+-------------+--------+
|     1|          120|     184|
|     3|           85|     355|
|     2|           83|     173|
+------+-------------+--------+
```

Como no nos importa ninguna columna en sí, a la hora de contar el número de filas utilizaremos el asterisco (*) como clave en lugar de un nombre de columna, de manera similar a como se hace en SQL. En caso de necesitar más flexibilidad también se pueden definir las distintas operaciones a realizar sobre los datos agregados mediante una secuencia de operaciones entre objetos `Column`. Se puede ver un ejemplo de este tipo de agregaciones en el *notebook* de este capítulo dentro del repositorio de código del libro. En todo caso, en la documentación del método `agg` se pueden encontrar más detalles sobre cómo expresar agregaciones complejas usando una secuencia de operaciones entre objetos `Column`.

SQL SOBRE DATAFRAMES

Para finalizar este apartado de operaciones sobre `DataFrames`, vamos a presentar una de las características más interesantes de SparkSQL para los usuarios de bases de datos relacionales. Hasta ahora hemos visto distintos métodos que nos permitían filtrar `DataFrames` (`filter`), combinar dos `DataFrames` (`union`, `join`, etc.), transformar columnas (`select` y `selectExpr`) o agrupar filas (`groupBy`). Todas estas operaciones son muy conocidas en el campo de las bases de datos relaciones, y de hecho SparkSQL se ha limitado a adaptarlas a su modelo de datos. Sin embargo, SparkSQL va un paso más allá y permite aplicar consultas SQL directamente sobre `DataFrames`.

Para poder ejecutar consultas SQL primero necesitaremos registrar los `DataFrames` involucrados para darles un nombre. Como los `DataFrames` son inmutables, registrarlos con un nombre será como definir una vista en una base de datos relacional. Para ello utilizaremos el método `createOrReplaceTempView` sobre el `DataFrame` que queremos registrar, y proporcionaremos el nombre deseado. Hay varios métodos dependiendo del rango de vida de la vista y de si queremos reemplazar una vista antigua o recibir una excepción en caso de que ya exista. En

este caso el método reemplaza la vista si existe previamente y esta tendrá la misma duración que el objeto `SparkSession` usado para crear el `DataFrame`. Por ejemplo, el siguiente código crea dos `DataFrames` y los registra con nombres *users* y *age*:

```
>>> users = spark.createDataFrame([(1,'ana'),(2,'jose')],
      ['id','nombre'])
>>> age = spark.createDataFrame([(1,36),(2,30)],['id','edad'])
>>> users.createOrReplaceTempView("users")
>>> age.createOrReplaceTempView("age")
```

A partir de este momento podemos realizar consultas SQL refiriéndonos a las tablas *users* y *age*. Esto nos permitiría combinar sus datos mediante una combinación interna (*inner join*) sobre su campo común *id*. Para ello únicamente tenemos que expresar la consulta SQL como una cadena de texto y pasarla como argumento de la función `spark.sql`:

```
>>> spark.sql("""SELECT *
                 FROM users INNER JOIN age
                 ON users.id == age.id""").show()
+---+------+---+----+
| id|nombre| id|edad|
+---+------+---+----+
|  1|   ana|  1|  36|
|  2|  jose|  2|  30|
+---+------+---+----+
```

Este es el mismo ejemplo que habíamos realizado de manera programática mediante el método `join` de los `DataFrames`. Internamente, SparkSQL procesará ambas alternativas y creará un plan de operaciones a ejecutar, que será equivalente independientemente de la manera elegida para expresar la consulta. Esto es una capacidad muy potente, ya que permite a cada programador expresar la consulta con la sintaxis que sienta más cómoda, mientras que la eficiencia, la distribución o la recuperación frente a fallos está garantizada por el sistema Spark subyacente.

Aparte de combinaciones de tablas, SparkSQL nos permite realizar consultas involucrando otras operaciones como transformaciones de columnas o agrupaciones. Una vez tenemos registrada la vista *titanic*, podemos filtrarla para conservar los pasajeros de más de 50 años y generar 3 columnas: la columna original *Survived*, la columna *Family* que suma las columnas *SibSp* y *Parch*, y la columna *Sex_Num* que transforma la columna *Sex* en un número entero aplicando la UDF *sex_to_num* que habíamos registrado en el apartado anterior:

```
>>> spark.sql("""SELECT Survived, SibSp+Parch AS Family,
                        sex_to_num(Sex) AS Sex_Num
                 FROM titanic
                 WHERE Age > 50""").show(3)
+--------+------+-------+
|Survived|Family|Sex_Num|
+--------+------+-------+
|       0|     0|      1|
|       1|     0|      0|
|       1|     0|      0|
+--------+------+-------+
```

SPARK ML

Spark dispone de un componente para realizar aprendizaje automático llamado MLlib. Como ya hemos comentado en el capítulo anterior, este componente proporciona dos interfaces de programación: a través de RDDs y a través de `DataFrames`. Como la interfaz que maneja RDDs está obsolescente y desaparecerá de las futuras versiones de Spark, en este capítulo nos centraremos únicamente en la interfaz de `DataFrames`, llamada comúnmente SparkML.

SparkML es muy similar a scikit-learn, y proporciona un amplio catálogo de clases dentro del paquete `pyspark.ml` para preprocesar conjuntos de datos, realizar clasificación, regresión, análisis de grupos, evaluar modelos, etc. Una diferencia notable con scikit-learn es que está íntegramente centrada en `DataFrames`. Scikit-learn permitía el uso de `DataFrames` de `pandas`, pero internamente trabajaba con objetos `ndarray` de NumPy y de hecho el resultado obtenido al realizar cualquier transformación era un `ndarray`. Esto puede ser un inconveniente si partíamos de un `DataFrame` de `pandas`, puesto que a partir de ese momento se perderían los nombres de columnas y cualquier operación se tendría que realizar usando los métodos de `ndarray`, que son menos potentes y sencillos que los de un `DataFrame` de `pandas`. Por el contrario, todas las clases de SparkML están diseñadas para aceptar `DataFrames` y devolver `DataFrames`, por lo que no hay ningún otro tipo de datos involucrado en el proceso y siempre se mantendrán los nombres de columnas.

Las clases proporcionadas por SparkML para realizar aprendizaje automático se pueden dividir en dos grupos. Por un lado tenemos los *transformadores*, que son clases que disponen del método `transform`. Este método recibe un `DataFrame` y devuelve otro `DataFrame` posiblemente extendido con algunas columnas. Un ejemplo de transformador es `VectorAssembler`, una clase que sirve para combinar un conjunto de columnas de un `DataFrame` y crear en una nueva columna con el vector resultante. Por otro lado, tenemos los *estimadores*, que son clases que

disponen del método `fit`. Este método recibe un `DataFrame` y devuelve un modelo, que será un objeto transformador. Un ejemplo de estimador es el clasificador `LinearSVC`, que al recibir un `DataFrame` a través de `fit` realiza el entrenamiento y genera un modelo que sirve para clasificar nuevas instancias a través de su método `transform`. Nótese que el comportamiento de `fit` en SparkML es ligeramente diferente al de scikit-learn, ya que en esa biblioteca los objetos que se podían adaptar con `fit` no devolvían ningún modelo, sino que lo almacenaban internamente, permitiendo invocar `transform` sobre el mismo objeto.

A la hora de realizar aprendizaje automático, las clases estimadoras recibirán un `DataFrame` a través de `fit` que puede tener cualquier número de columnas, pero por defecto esperarán una columna llamada `features` con todos los atributos que queremos considerar representados como un vector de números reales. El resto de columnas se ignorará. Esto también es una diferencia con respecto a scikit-learn, donde debíamos dividir los conjuntos de datos en dos (atributos y clase) y todo el contenido de estos conjuntos se utilizaba para el aprendizaje. En SparkML pueden aparecer muchos datos en el mismo `DataFrame`, pero a la hora de realizar aprendizaje automático deberemos aglutinar todos los atributos que queramos considerar en una única columna de vectores de números reales. Además, si el estimador realiza aprendizaje supervisado, también esperará por defecto una columna llamada `label` con un valor numérico representando la clase. De la misma manera, los modelos generados extenderán los `DataFrames` con una columna `prediction` por defecto al invocar a `transform`. Los nombres concretos utilizados por la columna de atributos, la columna clase o la columna de predicción se pueden configurar al crear los objetos estimadores.

Al igual que scikit-learn, SparkML también proporciona tuberías para facilitar el proceso del aprendizaje automático. Estas tuberías son secuencias encadenadas de estimadores y transformadores que finalizan con un estimador y que se utilizan de manera unificada. Al invocar al método `fit` de una tubería se invocará en orden a las distintas etapas: si la etapa es un transformador entonces se invoca al método `transform`, si la etapa es un estimador intermedio entonces se invoca al método `fit` seguido de `transform`, y si la etapa es el estimador final entonces únicamente se invoca al método `fit`. En todos los pasos intermedios se utilizará el `DataFrame` transformado para la siguiente etapa, de manera similar a las tuberías de scikit-learn. El resultado del método `fit` sobre una tubería será un `PipelineModel`, un objeto transformador que almacena los modelos intermedios y toda la información necesaria para poder transformar instancias nuevas en el futuro. Tanto las tuberías (objetos `Pipeline`) como los modelos de tubería (objetos `PipelineModel`) tienen un atributo `stages` que permite acceder a sus transformadores, estimadores y modelos.

En los próximos apartados veremos cómo crear y utilizar tuberías para realizar clasificación, regresión y análisis de grupos sobre el conjunto de datos sobre pasajeros del Titanic. También mostraremos cómo evaluar la calidad de los modelos generados, además de salvarlos y cargarlos.

Clasificación con SVM

En este apartado veremos cómo realizar una tubería para realizar clasificación sobre los pasajeros del Titanic usando SVM. A diferencia de lo realizado en el capítulo 6 con scikit-learn, donde los atributos nominales *Sex* y *Embarked* representados como cadenas de texto se traducían a números naturales antes de aplicar la tubería, en SparkML realizaremos ese proceso dentro de la propia tubería. El primer paso a realizar será cargar el DataFrame a partir del fichero, eliminar las columnas innecesarias, eliminar cualquier fila con valores vacíos y finalmente separar el conjunto en dos partes: 80% para entrenamiento y 20% para test:

```
>>> titanic = spark.read.csv('data/Cap8/titanic.csv', header=True,
                             inferSchema=True)
>>> titanic = titanic.drop('PassengerId', 'Name', 'Ticket','Cabin')
>>> titanic = titanic.dropna()
>>> train, test = titanic.randomSplit([0.8, 0.2])
```

El método `randomSplit` del `DataFrame` admite una lista de pesos que deben sumar 1 y fragmenta el `DataFrame` en tantas partes como pesos se hayan utilizado, cada una con un tamaño proporcional al peso.

Para crear la tubería de clasificación debemos crear tantos objetos como etapas vaya a tener, y configurarlos adecuadamente. Las primeras etapas se encargarán de traducir las columnas *Sex* y *Embarked* a números naturales, puesto que están representadas como cadenas de texto. Para ello utilizaremos la clase `StringIndexer` de la biblioteca `pyspark.ml.feature`. Al crear objetos de esta clase deberemos establecer el nombre de la columna de origen que contiene cadenas de texto (`inputCol`) y el nombre de la nueva columna que se creará para almacenar los números naturales (`outputCol`). A la hora de asignar números a cadenas de texto, por defecto `StringIndexer` los asigna por cantidad de apariciones de manera descendente. En este caso no es muy relevante los números asignados a cada cadena: en la columna *Sex* únicamente hay dos valores y su distancia siempre será 1, y la columna *Embarked* será transformada posteriormente mediante *one hot encoding*. A modo de ejemplo, en el siguiente código configuraremos los objetos `StringIndexer` para que asignen las etiquetas por orden alfabético ascendente mediante el parámetro `stringOrderType`. Este parámetro puede tomar los valores `frequencyDesc`, `frequencyAsc`, `alphabetDesc`, `alphabetAsc`.

```
>>> from pyspark.ml.feature import StringIndexer
>>> indexerSex = StringIndexer(inputCol='Sex', outputCol='Sex_num',
      stringOrderType='alphabetAsc')
>>> indexerEmbarked = StringIndexer(inputCol='Embarked',
      outputCol='Embarked_num', stringOrderType='alphabetAsc')
```

Como se puede ver, el primer objeto `indexerSex` generará una columna numérica *Sex_num* a partir de la columa *Sex*, mientras que el segundo objeto `indexerEmbarked` generará una columna numérica *Embarked_num* a partir de la columna *Embarked*. Aunque el proceso realizado en ambos casos es el mismo aplicado a distintas columnas, hemos necesitado crear dos objetos transformadores ya que no permite transformar varias columnas dentro del mismo objeto.

El siguiente paso será representar la columna recién creada *Embarked_num* usando la técnica *one hot encoding*. Para ello utilizaremos la clase `OneHotEncoderEstimator` de la biblioteca `pyspark.ml.feature`. Esta clase, a diferencia de `StringIndexer`, permite transformar varias columnas de golpe y generar su representación *one hot encoding*. Por ello recibe una lista de nombres de columnas de entrada (`inputCols`) y una lista de nombres de columnas a crear (`outputCols`). Estas listas deberán tener la misma longitud puesto que serán procesadas en orden. Una diferencia con scikit-learn es que `OneHotEncoderEstimator` genera exactamente una columna por cada columna de entrada, y cada columna generada almacenará vectores con tantas posiciones como sean necesarias. En el caso de *Embarked_num*, que tiene 3 posibles valores, la columna generada *Embarked_OHE* contendrá vectores de 3 posiciones como por ejemplo [0.0, 1.0, 0.0]. El siguiente código muestra la creación de esta etapa de la tubería:

```
>>> from pyspark.ml.feature import OneHotEncoderEstimator
>>> ohe = OneHotEncoderEstimator(inputCols=['Embarked_num'],
      outputCols=['Embarked_OHE'])
```

En este punto de la tubería ya dispondríamos de todos los atributos representados como números o como vectores de números en el caso de *Embarked_OHE*. Antes de proceder con la clasificación necesitaremos combinar todos los atributos en un único vector, puesto que las clases estimadoras de SparkML trabajan sobre una única columna. Para ello utilizaremos la clase `VectorAssembler` de la biblioteca `pyspark.ml.feature`. Esta clase es muy sencilla de configurar, únicamente necesita una lista con los nombres de las columnas a combinar (`inputCols`) y el nombre de la columna a crear (`outputCol`). Las columnas a fusionar en un único vector deben almacenar números o vectores de números. En nuestro caso vamos a combinar las columnas *Pclass*, *Sex_num*, *Age*, *SibSp*, *Parch*, *Fare* y

Embarked_OHE; donde todas son columnas numéricas salvo la última, que contiene un vector de números. El vector resultante lo almacenaremos en la nueva columna *features_raw*:

```
>>> from pyspark.ml.feature import VectorAssembler
>>> vec = VectorAssembler(inputCols=['Pclass', 'Sex_num', 'Age',
     'SibSp', 'Parch', 'Fare', 'Embarked_OHE'],
     outputCol='features_raw')
```

Antes de realizar la clasificación queremos escalar todos los atributos para que tomen un valor en el rango [0,1], al igual que hicimos en scikit-learn. Para ello utilizaremos la clase `MinMaxScaler` de la biblioteca `pyspark.ml.feature`. Esta clase recibe el nombre de la columna de origen (`inputCol`) y el nombre de la nueva columna a crear (`outputCol`). En nuestro caso queremos escalar la columna *features_raw* y generar una nueva columna *features*. Como *features_raw* contiene vectores, `MinMaxScaler` escalará cada posición de manera independiente:

```
>>> from pyspark.ml.feature import MinMaxScaler
>>> sca = MinMaxScaler(inputCol='features_raw', outputCol='features')
```

Finalmente, creamos la etapa de la tubería destinada a realizar clasificación. En este caso concreto usaremos la clase `LinearSVC` de la biblioteca `pyspark.ml.classification`, que realiza clasificación utilizando SVM. Para configurar este objeto tendremos que proporcionar el nombre de la columna con los atributos (`featuresCol`) y el nombre de la columna clase (`labelCol`). Este objeto también admite otros parámetros que configurarán el comportamiento del clasificador como el número máximo de iteraciones (`maxIter`, por defecto 100) o el umbral aplicado para la clasificación binaria (`threshold`, por defecto 0). En el ejemplo siguiente hemos tomado el valor por defecto en todos estos parámetros:

```
>>> from pyspark.ml.classification import LinearSVC
>>> clf = LinearSVC(featuresCol='features', labelCol='Survived')
```

Una vez que ya hemos definido todas etapas, el último paso será crear la tubería con todas las etapas en el orden adecuado. Para ello construiremos un objeto `Pipeline` de la biblioteca `pyspark.ml`:

```
>>> from pyspark.ml import Pipeline
>>> pipeline = Pipeline(stages=[indexerSex, indexerEmbarked, ohe,
                                vec, sca, clf])
```

A partir de este momento, el objeto `pipeline` actuará como un clasificador, es decir, se podrá entrenar para obtener un modelo. Para el entrenamiento utilizaremos

el conjunto de datos `train` que hemos creado anteriormente, y para comprobar la calidad del modelo usaremos el conjunto `test`. Al ejecutar el método `fit` sobre la tubería se entrenarán todas las etapas en orden y se generará un modelo de tipo `PipelineModel` que nos permitirá predecir las clases de conjuntos sin etiquetar.

```
>>> model = pipeline.fit(train)
>>> prediction = model.transform(test)
>>> prediction.select('prediction', 'Survived').show(5)
+----------+--------+
|prediction|Survived|
+----------+--------+
|       0.0|       0|
|       1.0|       1|
|       0.0|       0|
|       0.0|       0|
|       0.0|       1|
+----------+--------+
```

En el código anterior hemos generado el modelo `model` y lo hemos utilizado para clasificar las instancias del conjunto `test`. Para ello hemos invocado a `transform`, que devolverá un `DataFrame` similar a `test` pero con una columna adicional *prediction* con el valor de clase predicho para cada instancia. Como se puede comprobar al mostrar las 5 primeras instancias, el modelo ha acertado en todas menos una. Si lo que queremos es medir la calidad del modelo clasificador usando alguna de las métricas que vimos en el capítulo 6 sobre scikit-learn, deberemos utilizar un objeto evaluador `MulticlassClassificationEvaluator` de la clase `pyspark.ml.evaluation`. Este objeto tiene un método `evaluate` que recibe un `DataFrame` con la clase real y la predicha y calcula la métrica que le pidamos. Al crear este objeto deberemos seleccionar la columna con la clase real (`labelCol`), la columna con la clase predicha (`prediction`) y la métrica a utilizar (`metricName`):

```
>>> claseval = MulticlassClassificationEvaluator(
      predictionCol='prediction',
      labelCol='Survived',
      metricName='accuracy')
>>> print('Score:', claseval.evaluate(prediction))
Score: 0.7577639751552795
```

Es necesario indicar que, según la documentación de la versión 2.3.0 de Spark, las clases `LinearSVC` y `MulticlassClassificationEvaluator` se consideran experimentales. Esto es normal, puesto que se están portando funcionalidades del API de MLlib que usa RDDs al API que usa `DataFrames`. Sin embargo, en las próximas versiones de Spark desaparecerá este aviso.

Regresión lineal

Construir una tubería para aplicar regresión lineal es igual de sencillo que una tubería de clasificación. De hecho, las primeras 5 etapas serán iguales y únicamente cambiará la última. Para la etapa de regresión lineal utilizaremos la clase `LinearRegression` de la biblioteca `pyspark.ml.regression`. En este ejemplo tomaremos todos sus parámetros por defecto salvo la columna de atributos (`featuresCol`) y la columna con la clase (`labelCol`):

```
>>> reg = LinearRegression(featuresCol='features', labelCol='Fare')
>>> pipeline = Pipeline(stages=[indexerSex, indexerEmbarked, ohe,
                                vec, sca, reg])

>>> model = pipeline.fit(train)
>>> prediction = model.transform(test)
>>> prediction.select('Prediction', 'Fare').show(5)
+------------------+-------+
|        Prediction|   Fare|
+------------------+-------+
|103.39266593222533|  263.0|
| 93.20003777192207|82.1708|
|   69.888789208846|   66.6|
| 64.48334079746753|50.4958|
| 97.95400807625528|   29.7|
+------------------+-------+
```

Como se puede observar en la salida, la diferencia entre la predicción y la clase real difieren en cada instancia; por ejemplo, dista más de 100 libras en la primera, pero apenas 3 libras en la tercera. Si quisiéramos calcular una métrica como MAE, MSE o RMSE sobre el conjunto de test podemos utilizar una clase evaluadora de manera similar al caso de clasificación. En este caso utilizaríamos un objeto de la clase `RegressionEvaluator` de la biblioteca `pyspark.ml.evaluation`. En este caso crearemos 3 objetos evaluadores (`maeval`, `mseeval` y `rmseeval`), uno por cada métrica, e invocaremos a su método `evaluate` para obtener la métrica.

```
>>> maeeval = RegressionEvaluator(predictionCol='Prediction',
      labelCol='Fare', metricName='mae')
>>> print("MAE :", maeeval.evaluate(results))
MAE : 22.932462638797347
>>> mseeval = RegressionEvaluator(predictionCol='Prediction',
      labelCol='Fare', metricName='mse')
>>> print("MSE :", mseeval.evaluate(results))
MSE : 1417.8296133115746
>>> rmseeval = RegressionEvaluator(predictionCol='Prediction',
```

```
      labelCol='Fare', metricName='rmse')
>>> print("RMSE:", rmseeval.evaluate(results))
RMSE: 37.654078309149654
```

En este caso, al igual que para la clasificación, el objeto evaluador `RegressionEvaluator` está considerado experimental en la versión 2.3.0.

Análisis de grupos con k-means

Para realizar análisis de grupos mediante una tubería seguiremos los mismos pasos que para la clasificación o regresión, pero cambiando la última etapa de la tubería. En este caso usaremos la clase `KMeans` de la biblioteca `pyspark.ml.clustering` para realizar un análisis de grupos siguiendo la técnica k-means. Para construir esta etapa usaremos los parámetros por defecto: la columna con los atributos se llamará *features* y la predicción, es decir, el clúster al que se asigna cada instancia, se almacenará en una nueva columna de nombre *prediction*. Sí que deberemos configurar el número de grupos que queremos formar a través del parámetro `k`, en este caso 3:

```
>>> clu = KMeans(k=3)
>>> pipeline = Pipeline(stages=[indexerSex, indexerEmbarked, ohe,
      vec, sca, clu])


>>> model = pipeline.fit(titanic)
>>> prediction = model.transform(titanic)
>>> prediction.select('prediction').show(5)
+----------+
|prediction|
+----------+
|         0|
|         2|
|         2|
|         2|
|         0|
+----------+
```

Obsérvese cómo en este caso hemos realizado el entrenamiento de la tubería con todas las instancias disponibles en `titanic`, puesto que no disponemos de un atributo clase que nos pueda servir para evaluar la calidad del modelo. Una vez obtenido el modelo, usamos su método `transform` sobre el mismo conjunto `titanic` para que añada una columna *prediction* con el número de clúster, que en este caso podrá tomar valores 0, 1 y 2 puesto que hemos configurado `k` a 3.

Si además del número de clúster nos interesa conocer los centroides concretos, podemos acceder a esa información a través de `model`. Como se trata de un `PipelineModel`, dispondrá de un atributo `stages` con una lista los modelos obtenidos para etapa, donde el último será el modelo de la etapa de análisis de grupos con k-means. Este modelo a su vez tiene un método `clustersCenters` que devuelve una lista con los valores de los 3 centroides, cada uno representado como un objeto `ndarray` de NumPy:

```
>>> for c in model.stages[-1].clusterCenters():
      print(c)
[0.7575406  0.87006961 0.37345076 0.09791183 0.05800464 0.06960557
 0.11600928 0.         0.04299367]
[0.92592593 0.59259259 0.34540597 0.14814815 0.06790123 0.25925926
 0.         1.         0.03046664]
[0.35433071 0.24409449 0.35892125 0.10629921 0.09645669 0.98818898
 0.31496063 0.00393701 0.11293829]
```

Finalmente, para la evaluación de la calidad del análisis de grupos, SparkML proporciona en la biblioteca `pyspark.ml.evaluation` una clase llamada `ClusteringEvaluator` (experimental en Spark 2.3.0) que nos permite calcular métricas de calidad de la agrupación generada. Actualmente solo soporta la métrica `"silhouette"` para calcular el coeficiente de silueta, aunque es muy posible que en el futuro se añadan más métricas según la clase sea más estable. Para crear este objeto evaluador debemos configurar la columna con la predicción (`predictionCol`), la columna con los atributos (`featuresCol`) y la métrica a utilizar (`metricName`), aunque en este caso utilizaremos todos los valores por defecto:

```
>>> evaluator = ClusteringEvaluator()
>>> print('Silhouette Coefficient:', evaluator.evaluate(prediction))
Silhouette Coefficient: 0.5096854350632269
```

Persistencia de modelos

De manera similar al caso de scikit-learn, SparkML nos permite almacenar los modelos generados como resultado de entrenar una tubería. De esta manera podremos guardar un modelo, cuyo tiempo de entrenamiento puede haber sido elevado, y recuperarlo en el futuro para realizar predicciones.

La manera más sencilla de almacenar un modelo de tubería (objeto `PipelineModel`) es invocar a su método `save`, que recibe una ruta donde almacenar el modelo. Esta ruta puede ser local o remota, al igual que ocurría con los RDDs y los `DataFrames`. De manera similar a estas estructuras de datos, al volcar un modelo de tubería a disco se creará una carpeta en lugar de un fichero. Esta carpeta contendrá

varias subcarpetas conteniendo la configuración del modelo y la representación de cada etapa. Retomando la tubería para realizar regresión lineal vista anteriormente, almacenar el modelo generado en la carpeta `data/Cap8/regression_model` sería tan sencillo como invocar al método `save`:

```
>>> pipeline = Pipeline(stages=[indexerSex, indexerEmbarked, ohe,
                                vec, sca, reg])
>>> model = pipeline.fit(train)
>>> model.save('data/Cap8/regression_model')
```

Esta invocación supone que la carpeta no existe, o lanzará un error. Para evitar esta situación, se puede realizar el volcado con sobrescritura a través del objeto `MLWriter` que devuelve el método `write`:

```
>>> model.write().overwrite().save('data/Cap8/regression_model')
```

Una vez se tiene un modelo almacenado en disco, recuperarlo únicamente requiere invocar al método `load` de la clase `PipelineModel`. El objeto cargado es un modelo con exactamente la misma funcionalidad que el modelo original generado por `fit`, y por tanto servirá para transformar `DataFrames` para añadir la predicción:

```
>>> loaded_model = PipelineModel.load('data/Cap8/regression_model')
>>> prediction = loaded_model.transform(test)
>>> prediction.select('Prediction', 'Fare').show(5)
+------------------+-------+
|        Prediction|   Fare|
+------------------+-------+
|103.39266593222533|  263.0|
| 93.20003777192207|82.1708|
|   69.888789208846|   66.6|
| 64.48334079746753|50.4958|
| 97.95400807625528|   29.7|
+------------------+-------+
```

REFERENCIAS

- Learning Spark: Lightning-fast Data Analysis. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. O'Reilly, 2015.
- High Performance Spark: Best Practices for Scaling & Optimizing Apache Spark. Holden Karau, Rachel Warren. O'Reilly, 2017.
- Página principal de Apache Spark: `https://spark.apache.org/`

- Documentación de la clase `DataFrame`: `https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame`
- Deep Dive into Spark SQL's Catalyst Optimizer. Michael Armbrust, Yin Huai, Cheng Liang, Reynold Xin, Matei Zaharia. Blog de DataBricks `https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html`
- Spark SQL: Relational Data Processing in Spark. Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia. SIGMOD Conference 2015: 1383-1394.
- MongoDB Connector for Spark: `https://docs.mongodb.com/spark-connector/master/`
- Spark Connector Python Guide: `https://docs.mongodb.com/spark-connector/master/python-api/`

8 VISUALIZACIÓN DE RESULTADOS

INTRODUCCIÓN

En los capítulos anteriores hemos aprendido a obtener datos de la red, a cargarlos en Python y a manipularlos de diversas maneras. Pero ¿por qué tanto esfuerzo? Porque en la sociedad actual aquel que es capaz de extraer información de los datos en bruto puede tomar decisiones mejor informadas y obtener una ventaja respecto al resto. El problema ahora consiste en cómo interpretar nuestros resultados, y a continuación en cómo lograr transmitir nuestras conclusiones a los demás. La solución está en el dicho *una imagen vale más que mil palabras*, la representación gráfica de nuestros resultados nos permitirá entender los datos y tomar decisiones en relación con ellos.

En este capítulo presentamos la biblioteca `matplotlib`, que nos permite representar una amplia variedad de diagramas para visualizar nuestros datos. Empezaremos explicando brevemente la arquitectura de la biblioteca y cómo representar funciones para presentar a continuación gráficas circulares, gráficas de caja, gráficas de barras e histogramas.

LA BIBLIOTECA MATPLOTLIB

El módulo principal en el módulo `matplotlib` es `pyplot`, que define las clases para figuras y ejes. Por ello, la supondremos cargada como:

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

Intuitivamente, una figura (clase `Figure`) es un conjunto de ejes (clase `Axes`). Cada uno de estos "ejes" es lo que nosotros entendemos intuitivamente como una "gráfica". Es decir, en Python crearemos la imagen presentada en la figura 8-1.

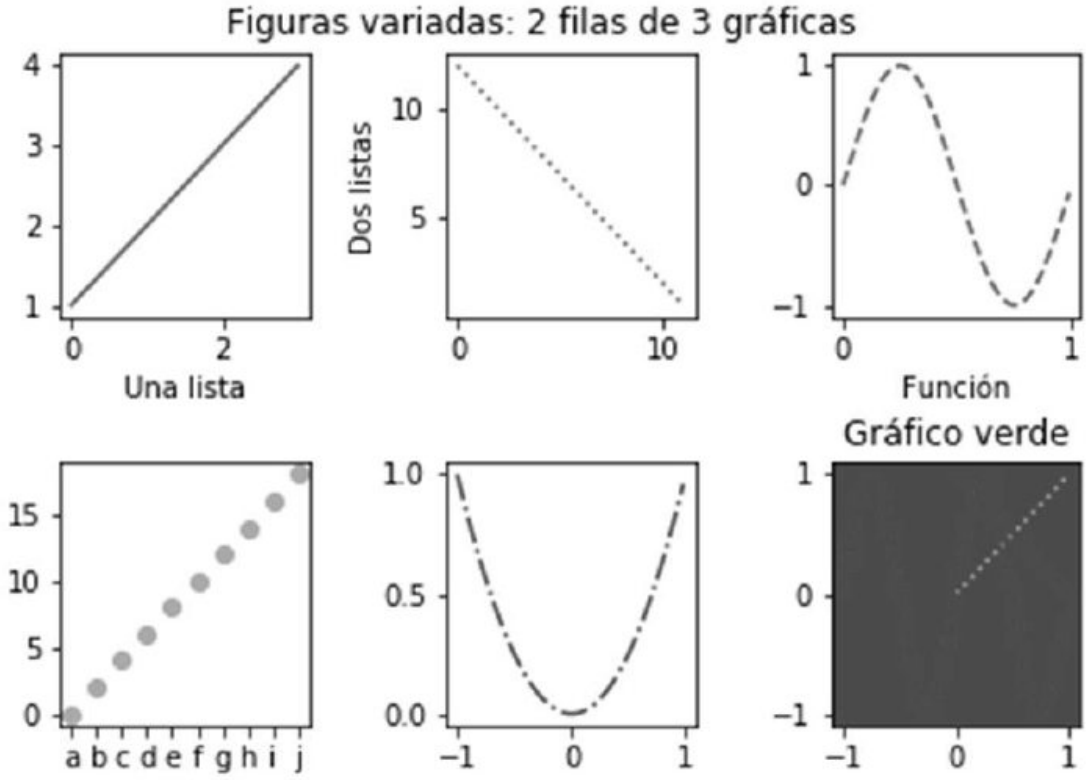


Figura 8 -1. Ejemplo de figura usando matplotlib.

Entendemos esta imagen como una sola figura que contiene 6 ejes. Los ejes, a su vez, se componen de 2 *objetos eje* (clase `Axis`), o de 3 en el caso de las figuras en 3 dimensiones. Podemos trabajar de dos maneras con la biblioteca `matplotlib`: sobre figuras/ejes particulares o sobre el módulo `pyplot`, que delegará las llamadas a los objetos actualmente en uso. En este capítulo usaremos la segunda aproximación, por evitar las distinciones entre figuras y ejes y tener una presentación más homogénea. Las funciones principales del módulo `pyplot` son:

- La función `subplots`, que toma en general dos argumentos `n` y `m` indicando el número de filas y columnas que tendrá la figura, es decir, indicando que tendremos `n` filas con `m` ejes cada una de ellas. Esta función cambia el foco actual a la figura creada y a los últimos ejes (los situados en la posición inferior derecha). En caso de querer crear una figura con unos únicos ejes podemos llamar a la función sin pasarle ningún argumento. El formato usado para devolver los objetos creados es una pareja en la que el primer objeto es la figura y el segundo es una lista, donde cada elemento es, a su vez, una lista

que corresponde a una fila de ejes. Es decir, el segundo argumento es una lista con `n` listas, cada uno de `m` objetos de tipo Axes.

- La función `tight_layout` ajusta el tamaño de los ejes para asegurarse de que todas las etiquetas se muestran y no se solapan distintos ejes. Esta función consigue buenos resultados en la mayoría de los casos, pero en caso de fallar podemos usar la función `subplots_adjust`, que nos permite ajustar las distancias en la figura. Esta función recibe como argumentos los siguientes valores, todos ellos números reales entre 0 y 1:
  - `left` y `right`, que indican la fracción de anchura dedicada a espacio en blanco a la izquierda y la derecha, respectivamente, de la figura.
  - `top` y `bottom`, que indican la fracción de altura dedicada a espacio en blanco encima y debajo, respectivamente, de la figura.
  - `wspace`, que indica la fracción del ancho total de la figura dedicado a separación horizontal entre figuras.
  - `hspace`, que indica la fracción de la altura total de la figura dedicada a separación vertical entre figuras.
- La función `subplot` cambia el foco a los ejes dados como argumento. En particular, es necesario darle como argumentos el número de filas y el número de columnas que tenemos en la imagen actual y el índice de los ejes, donde el índice 1 corresponde a los ejes situados en la posición superior izquierda y se incrementa de izquierda a derecha y de arriba abajo. Así, dada la figura 8.1, con 2 filas de 3 columnas la llamada `subplot(2,3,3)` devuelve la imagen superior derecha y `subplot(2,3,4)` la inferior izquierda. Cuando los dígitos necesarios para la llamada son todos menores que 10 es habitual usar la función con una sola cifra de 3 dígitos obtenida al juntar todos los argumentos; por ejemplo, la llamada `subplot(233)` es equivalente a `subplot(2,3,3)`. Es interesante notar que es posible usar un número de filas o columnas diferente del que usamos al crear la figura; los ejes correspondientes se crearán en la posición indicada en el supuesto de que la figura tuviese la disposición indicada y eliminará aquellos ejes que "pise". Así, puede utilizarse, por ejemplo, para crear una figura 2 x 2 que tenga dos imágenes en la fila superior pero solo una, que ocupe tanto como las superiores, en la fila inferior. Para ello, primero habría que crear una figura de cuatro ejes 2 x 2 con `subplots(2,2)`, y luego sobrescribir la fila inferior (que contendría dos ejes) con un único eje. Eso se realizaría "reinterpretando" la figura como si tuviera tamaño 2 x 1 (2 filas y una única columna) y accediendo a sus segundos ejes (la fila inferior) con `subplot(2,1,2)`.
- Las funciones `gcf` y `gca` (*getcurrentfigure* y *getcurrentaxes*) devuelven la figura y los ejes actuales, respectivamente. Análogamente, las funciones scf y

sca (*setcurrentfigure* y *setcurrentaxes*) reciben como argumentos una figura y unos ejes, respectivamente, y los fijan como actuales.

- La función `suptitle` fija el título de la figura actual, que se sitúa centrado en la parte superior de la figura.
- La función `title` fija el título de los ejes actuales, situándolo centrado en la parte superior.
- La función `xlabel` modifica la etiqueta en el eje horizontal de los ejes actuales. Por su parte, la función `ylabel` modifica el eje vertical de los ejes actuales.
- La función `xticks` sirve para acceder y modificar las marcas en el eje horizontal. Si lo usamos sin argumentos devolverá una pareja con dos listas, la primera indicando las posiciones del eje en las que habrá marcas, y la segunda indicando la información que se mostrará en dichas marcas. Si queremos modificar estos datos, usaremos la misma función, pero pasándole como argumentos dos listas con esta información. En este caso la primera lista será de números, mientras la segunda será de cadenas. Un modo especial de usar la función es con una sola lista vacía, lo que elimina las marcas y la correspondiente información. La función `yticks` funciona de manera análoga para el eje vertical.
- La función `axis` sin argumentos devuelve una lista `[xmin,xmax,ymin,ymax]` con los límites de la gráfica. Sin embargo, si le pasamos argumentos podemos usarla para modificar sus límites. Algunos valores interesantes que podemos pasarle son:
    - `axis(l)`, con `l` una lista de cuatro elementos indicando los límites de la gráfica como se mostró arriba, actualiza los límites.
    - `axis('off')`, que elimina los ejes y las etiquetas.
    - `axis('equal')`, que indica que iguales incrementos en los ejes tienes la misma longitud. Es la opción usada para asegurarnos de que los círculos se dibujan como círculos y no como elipses.
    - `axis('scaled')`, que consigue el mismo efecto que la función anterior, pero modificando la caja que contiene a la gráfica en lugar de la longitud de los ejes.
    - `axis('tight')`, que se asegura que todos los valores se muestran.
- La función `plot` dibuja una gráfica en los ejes actuales. Esta función permite el uso de muchas variantes, que merece la pena explicar en detalle:
    - Si le pasamos una lista de n números como argumento e staremos indicando la componente y de una gráfica con n puntos que tienen como componente x los elementos de la lista [0, ..., n-1]. Así, si hacemos la llamada `plot([1, 2, 4, 8])` estaremos creando la curva que pasa por los puntos (0, 1), (1, 2), (2,4) y (3, 8).

- Si le pasamos como argumento dos listas de n números cada una, la primera lista indica las componentes y de la gráfica y la segunda lista las correspondientes componentes x. Por ejemplo, la llamada `plot([1,2,3,4], [0,1,2,3])` replica la llamada explicada en el punto anterior.
- Podemos usar la función `arange` de la biblioteca `numpy` para generar un rango de datos al que aplicar funciones. En este caso, pasaríamos el rango como primer argumento y la función como segundo. Por ejemplo, podemos dibujar una parábola entre los puntos -1 y 1, tomando valores con distancia 0.01 (lo que haremos con `arange(-1,1,0.01)`, que supondremos almacenado en la variable `x`), con la llamada a función `plot(x, x**2)`.
- Por defecto, las gráficas se muestran con una línea continua de color azul. Podemos cambiar el color de la línea si le pasamos como argumento a la función un carácter que indique el nuevo color. Este carácter suele ser la primera letra del nombre del color en inglés, con lo que `'g'` hará que dibujemos una línea verde, `'y'` una línea amarilla, etc. Asimismo, podemos usar una cadena para variar el estilo de la línea, los más utilizados son:
  - `'--'` para líneas discontinuas. Podemos usar también el valor `'dashed'`.
  - `':'` para líneas de puntos. Podemos usar también el valor `'dotted'`
  - `'.-'` para líneas discontinuas formadas por puntos y guiones. Es equivalente al uso del valor `'dashdot'`.

  Si en lugar de rectas queremos crear la gráfica con otros marcadores deberemos usar otros caracteres como argumento. Los más habituales son:

  - `'o'` para círculos.
  - `'*'` para estrellas.
  - `'p'` para pentágonos.
  - `'D'` y `'d'` para diamantes y diamantes estrechos, respectivamente.

  Se pueden combinar los 3 elementos en un solo argumento, indicando primero el color, luego el marcador y, por último, el estilo de la línea (en este caso solo usando las versiones abreviadas). Por ejemplo, la combinación `'go'` indica que se debe dibujar una línea verde con círculos y la combinación `'r--'` indica una línea roja discontinua. También es posible indicar el color, el estilo y el

marcador de la línea con los parámetros `color`, `marker` y `linestlye`. Así, el modificador `'r--'` puede escribirse con los argumentos `color='r', linestyle='--'`. El listado completo de posibles colores, marcadores y estilos se indica en las referencias.

- Es posible dibujar varias gráficas con una sola llamada a la función `plot`. Para ello, basta con usar como argumentos todos aquellos valores que serían necesarios para mostrar las gráficas. Por ejemplo, supongamos que queremos dibujar la recta `y = x` en rojo y la recta `y = -x + 1` en verde, ambas entre los valores 0 y 2 Para ello escribiríamos `plot([0,1,2], 'r', [1,0,-1], 'g')`.

- La función `savefig` guarda la figura actual un fichero con formato PNG en la ruta pasada como parámetro.
- La función `show` muestra la figura actual.

Veamos cómo usar estas funciones para crear los elementos que mostramos en la figura 8-1. Empezaremos por usar `subplots` para crear una figura con 6 ejes, 3 por fila. Guardaremos los objetos creados, observando que el segundo valor es una lista de que contiene 2 listas, una por fila.

```
>>> fig, [[arriba_iz, arriba_cnt, arriba_dr],
          [abajo_iz, abajo_cnt, abajo_dr]] = plt.subplots(2,3)
```

Una vez creada la figura, ajustamos las dimensiones con `subplots_adjust` para no tener elementos superpuestos con otros y fijamos el título con `suptitle`.

```
>>> plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95,
                        hspace=0.55, wspace=0.55)
>>> plt.suptitle('Figuras variadas: 2 filas de 3 gráficas')
```

Empezamos con los ejes situados en la parte superior izquierda, en los que simplemente dibujaremos una recta usando la función `plot` con una sola lista; las coordenadas del eje horizontal se generarán automáticamente desde 0. Por último, ponemos como etiqueta del eje horizontal `'Una lista'`:

```
>>> plt.sca(arriba_iz)
>>> plt.plot([1,2,3,4])
>>> plt.xlabel('Una lista')
```

Construiremos la gráfica situada en mitad de la fila superior con dos listas. Para ello, usamos la función predefinida `range`, ascendente en el primer caso y descendente en el segundo, obteniendo una recta descendente. Además, indicamos

que la recta será punteada dando el valor 'dotted' a linestyle. En este caso asignamos la etiqueta 'Dos listas' al eje vertical.

```
>>> plt.sca(arriba_cnt)
>>> plt.plot(range(12), range(12, 0, -1), linestyle='dotted')
>>> plt.ylabel('Dos listas')
```

Para la imagen en el extremo derecho de la fila superior dibujaremos la gráfica para la función seno entre 0 y 1. Para ello suponemos cargada la biblioteca numpy como np y usaremos arange para crear un rango de valores con una distancia 0.01 entre elementos consecutivos. Usaremos entonces la función sin para obtener el valor del seno de 2πx y usaremos la opción dashed para obtener una línea discontinua. Por último, el eje horizontal tendrá la etiqueta 'Función'.

```
>>> plt.sca(arriba_dr)
>>> x = np.arange(0.0, 1.0, 0.01)
>>> plt.plot(x, np.sin(2*np.pi*x), linestyle='dashed')
>>> plt.xlabel('Función')
```

El gráfico en el extremo izquierdo de la fila inferior dibuja la recta usando las opciones que hemos visto en los ejes anteriores, pero además renombra las marcas que aparecen en el eje horizontal y pone en su lugar las letras de 'a' hasta 'j'.

```
>>> plt.sca(abajo_iz)
>>> plt.plot(range(10), range(0,20,2), 'yo')
>>> plt.xticks(range(10), ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i', 'j'])
```

En la gráfica en el centro de la fila inferior dibujamos una parábola roja con puntos y guiones:

```
>>> plt.sca(abajo_cnt)
>>> x = np.arange(-1.0, 1.0, 0.02)
>>> plt.plot(x, x**2, 'r',linestyle='dashdot')
```

Por último, dibujaremos varias curvas en los ejes situados en el extremo derecho de la fila inferior. Cambiaremos el foco a estos ejes con subplot, indicando que queremos el fondo verde. Entonces, dibujaremos dos rectas que se crucen entre 0 y 1 y la función seno entre -1 y 1 (por lo que las rectas quedarán en la esquina superior derecha).

```
>>> plt.subplot(236, facecolor='g')
>>> plt.plot([0,1], "y:", [1,0], "r", x, np.sin(2*np.pi*x), "m--")
>>> plt.title('Gráfico verde')
```

Una vez modificadas todas las gráficas, guardamos la imagen en la carpeta local y la mostramos.

```
>>> plt.savefig("figuras_variadas")
>>> plt.show()
```

GRÁFICAS

Además de la función `plot`, `matplotlib` proporciona funciones para representar gráficas de manera sencilla, básicamente pasándoles como parámetro una lista con los datos que queremos representar. Las gráficas que veremos en esta sección son las gráfica circular, de caja y de barras, así como histogramas.

Para ejemplificar estas gráficas usaremos los datos de desempleo proporcionados para el mes de marzo de 2018 por el Ministerio de Empleo y Seguridad Social, disponibles en:

`https://www.sepe.es/contenidos/que_es_el_sepe/estadisticas/datos_avance/paro/index.html`

así como en el repositorio asociado a este libro. Por ello, a lo largo de la sección consideraremos cargada la variable `ruta_datos` como:

```
>>> ruta_datos = "data/Cap8/parosexoedadprov.xls"
```

El fichero tiene formato XLS, se lee como indicamos en el capítulo 1 y contiene información sobre el paro por provincias y comunidades autónomas, distinguiendo por sexo y por los tramos de edad: menores de 25 años, entre 25 y 29 años, entre 30 y 44 años y mayores de 45 años. Todas las gráficas precisan procesar de distintas maneras este fichero; los lectores que han llegado hasta este capítulo tienen ya el nivel suficiente para esta tarea, por lo que no mostraremos las funciones auxiliares. En todo caso, los lectores interesados pueden comparar sus soluciones con el código proporcionado en el repositorio.

Gráfica circular

La gráfica circular, también llamada de tarta, se usa para representar proporciones sobre una cierta población como secciones circulares. `matplotlib` proporciona la función `pie` para crear estas gráficas, que acepta los siguientes parámetros:

- `x`, la lista de valores que se mostrarán. Para cada valor se mostrará el área correspondiente al valor de x respecto a la suma total de valores en la lista. Además, si el valor de la suma es inferior a 1 la gráfica tendrá una parte sombreada, indicando que el total no llega al 100%. Solo este argumento es obligatorio.
- `labels`, la lista de etiquetas que dan nombre a cada uno de los valores.
- `explode`, una lista de valores entre 0 y 1 que indican la fracción de radio que cada porción de la gráfica se separa del centro del círculo.
- `autopct`, un *string* o una función que indica cómo mostrar el porcentaje, siguiendo los criterios de formato de Python (ver referencias para más detalles).
- `shadow`, booleano que indica si la gráfica debe tener efecto de sombra.
- `startangle`, un número real entre 0 y 360 que indica dónde empieza la primera sección. Por defecto esta sección empieza en 0 grados y `startangle` indica la desviación en sentido antihorario, por lo que es típico usar el valor 90 para empezar la sección en la vertical, lo que produce un efecto visual agradable.

Esta función devuelve una terna con 3 listas, la primera es una lista de objetos de clase `Wedge`, que son las distintas "cuñas" en las que se ha dividido el círculo, la segunda son los textos usados para cada una de estas cuñas y la tercera son los textos usados para los porcentajes. Cuando dibujamos este tipo de gráfica es interesante usar la función `legend`, que recibe como parámetros las cuñas de la gráfica, los textos que queremos usar para cada una de ellas y su localización (recomendamos usar la localización `"best"`, que automáticamente busca el mejor lugar para situar el correspondiente cuadro).

Para ilustrar este tipo de gráfica en esta sección implementaremos la función `paro_edades_tarta`, que muestra los porcentajes de paro en una ciudad concreta y distinguiendo entre hombres y mujeres. Para ello suponemos la existencia de una función `calcula_estadisticas` que recibe como parámetro el nombre de una ciudad, la ruta del fichero con las estadísticas y un booleano indicando si queremos datos para las mujeres (`True`) o para los hombres (`False`) y devuelve una pareja de listas. La primera es la lista con los valores extraídos del fichero, mientras que la segunda es una lista de etiquetas indicando las categorías. Así, una llamada a la función para las mujeres de Almería sería:

```
>>> calcula_estadisticas("ALMERIA", ruta_datos, True)
([2435.0, 3744.0, 12567.0, 13774.0], ['MENORES DE 25 AÑOS',   'DE 25
A 29 AÑOS','DE 30 A 44 AÑOS','MAYORES DE 45 AÑOS'])
```

Nuestra función para generar la gráfica empieza llamando a esta función auxiliar y usando `subplots` para crear unos nuevos ejes. Definimos una tupla para indicar que todas las categorías aparecerán pegadas al centro del círculo excepto la última, que estará separada un 10% del tamaño del radio. Usamos entonces la función `pie` para crear la gráfica, usando los valores definidos anteriormente e indicando que los porcentajes se mostrarán con un solo decimal (`autopct`), con sombras (`shadow`) y que se empezará generando secciones en el eje y (`startangle`). Además, nos aseguramos de que los ejes se muestren iguales (`axis`) y presentamos la leyenda en la mejor posición posible.

```
def paro_edades_tarta(ciudad, ruta, mujeres=True):
    vals, etqs = calcula_estadisticas(ciudad, ruta, mujeres)

    fig, ejes = plt.subplots()

    explode = (0, 0, 0, 0.1)
    cunhas, text_cat, text_porc = ejes.pie(vals, explode=explode,
labels=etqs, autopct='%1.1f%%', shadow=True, startangle=90)
    ejes.axis('equal')
    plt.legend(cunhas, etqs, loc="best")
    plt.show()
```

Si comprobamos, por ejemplo, los valores obtenidos para la provincia de Almería:

```
>>> paro_edades_tarta("ALMERIA", ruta_datos)
```

obtenemos la figura mostrada en la figura 8-2. Observamos cómo se plasman en la gráfica las opciones definidas en el código sobre sombras, separación y colocación de las distintas secciones. Si decidimos analizar los resultados, observamos que el paro es mayor en los dos últimos tramos, lo que parece en parte razonable, porque se refieren a un rango de valores más grande y, además, a unas edades en las que es menos habitual estar estudiando. Analizaremos si esta observación es correcta con una gráfica de caja.

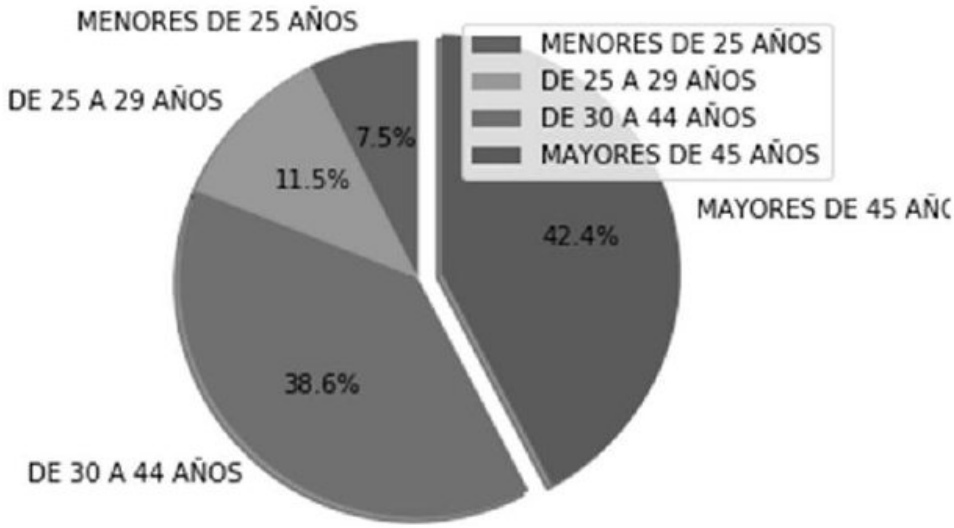


Figura 8-2. Gráfica circular con el paro en Almería.

Gráfica de caja

En la gráfica de caja (*boxplot*) podemos mostrar información sobre la mediana y los distintos cuartiles en los que aparecen los datos. Esta gráfica en general muestra un rectángulo, que contiene el 50% de los datos (aquellos comprendidos en los dos cuartiles centrales) y en particular la mediana, y unos brazos o "bigotes" que llegan hasta el máximo y el mínimo, siempre que estos se encuentren a una distancia de menos de 1,5 el rango intercuartílico del correspondiente extremo del rectángulo. A los valores más alejados de esta distancia se les llama valores atípicos o *outliers* y se representan por separado.

La biblioteca `matplotlib` ofrece la función `boxplot` para generar diagramas de caja, que recibe los siguientes parámetros:

- `x`, una lista de valores o una lista de listas de valores. Si es una lista de valores generará una sola caja, mientras que si es una lista de listas generará tantas cajas como listas. Solo este argumento es obligatorio.
- `notch`, booleano (por defecto `False`) que indica si el rectángulo debe tener una "muesca" indicando más visualmente la mediana.

- `sym`, un *string* que indica el símbolo utilizado para los valores atípicos. Por defecto es un círculo.
- `vert`, booleano (por defecto `True`) que indica si las cajas se muestran en vertical (`True`) o en horizontal (`False`).
- `whis`, real (por defecto 1,5) que indica el valor a multiplicar por el rango intercuartílico para aceptar valores máximos y mínimos.
- `usermedians`, lista de valores compatible con `x` que sobrescribe las medianas calculadas por Python. Si alguno de los valores es `None` se usa el valor computado por Python.
- `positions`, una lista de enteros, de longitud compatible con `x`, que indica las posiciones en las que se situarán las cajas (por defecto su valor es 1, ..., n, con n la cantidad de cajas a mostrar).
- `widths`, una lista de reales, de longitud compatible con `x`, que indica el ancho de las cajas (por defecto 0,5 o, si no hay suficiente espacio, 0,15 multiplicado por la distancia entre extremos).
- `labels`, una lista de cadenas de texto, de longitud compatible con `x`, que indica las etiquetas utilizadas para designar las cajas.

Esta función devuelve un diccionario que contiene listas de objetos gráficos. En particular, el diccionario tiene las claves:

- `boxes`, con información sobre las cajas del diagrama y los intervalos de confianza.
- `medians`, con las líneas que representan la mediana de cada caja.
- `whiskers`, con los brazos de la caja, que llegan hasta los valores mínimo y máximo.
- `caps`, con las líneas que denotan los valores mínimo y máximo.
- `fliers`, con los valores atípicos.
- `means`, con los valores de las medias.

Vamos a usar un diagrama de caja para visualizar cómo varía el porcentaje de parados por franjas de edad en todas las provincias españolas. Para ello usamos la función auxiliar `datos_provincia_porcentajes` que, dada la ruta del fichero, calcula una lista de cuatro listas. Cada una de estas listas contiene los porcentajes para todas las ciudades en un cierto rango de edad. Así, la primera lista contiene los porcentajes de paro para los menores de 25 años en todas las provincias, la segunda lista los porcentajes para las personas con edades comprendidas entre 25 y 29 años, etcétera. Así, la llamada con nuestros datos devuelve una lista con 4 listas, cada una de ellas con 53 elementos:

```
>>> datos_provincia_porcentajes(ruta_datos)
[[...], [...], [...], [...]]
```

La función `diagrama_caja` se encarga de dibujar la correspondiente gráfica, para lo que recopila la información calculada por la función auxiliar, crea unos nuevos ejes, una lista de etiquetas (para asegurarnos de que caben en el diagrama) y llama a la función `boxplot` con los datos y las etiquetas:

```
def diagrama_caja(ruta):
    totales = datos_provincia_porcentajes(ruta)
    plt.subplots()
    etqs = ["Menores 25", "25-29", "30-44", "Mayores 45"]
    plt.boxplot(totales, labels=etqs)
    plt.show()
```

Si hacemos la llamada con la ruta de datos definida para nuestro fichero:

```
>>> diagrama_caja(ruta_datos)
```

Obtenemos el diagrama en la figura 8-3. En él observamos que los valores que obtuvimos para Almería en el apartado anterior son razonablemente parecidos en el resto de España, con cajas estrechas y pocos valores atípicos.

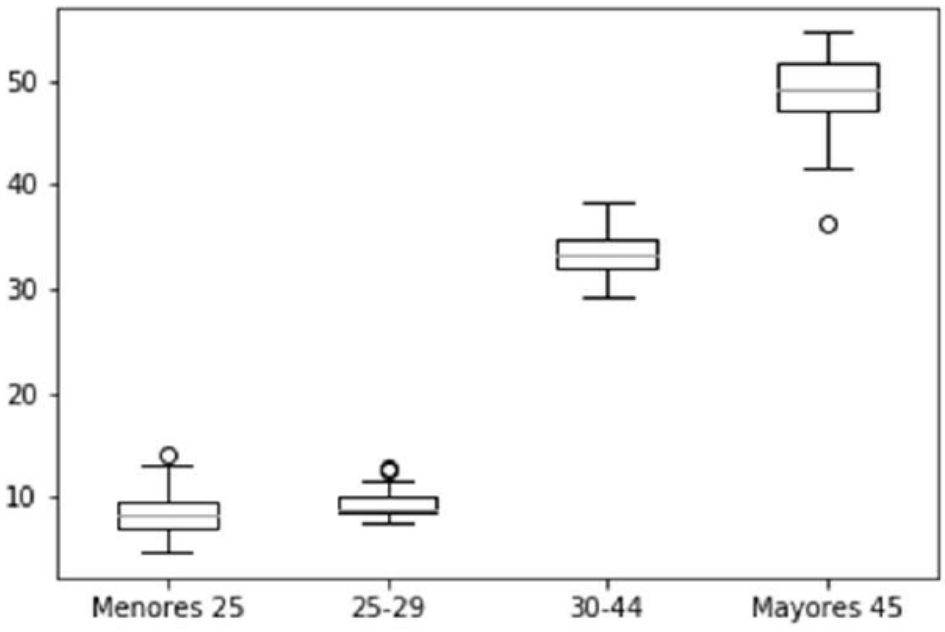


Figura 8-3. Diagrama de caja con información del paro por franjas de edad.

En las próximas secciones veremos cómo crear diagramas de barras e histogramas para recopilar más información por ciudades.

Gráfica de barras

Las gráficas de barras presentan barras verticales proporcionales a los valores representados. Esta representación permite visualizar fácilmente cantidades y hacer comparaciones entre ellas.

La biblioteca `matplotlib` proporciona la función `bar` para generar estas gráficas. Esta función recibe los siguientes parámetros:

- `x`, una secuencia de posiciones en el eje horizontal donde se situarán las barras. Este argumento es obligatorio.
- `height`, un valor o lista de valores con la altura de las barras. Este argumento es obligatorio.
- `width`, un valor o lista de valores que indica la anchura de las barras. Su valor por defecto es 0,8.
- `bottom`, un valor o lista de valores que indica la posición en el eje y en la que empieza cada una de las barras.
- `align`, que puede tomar los valores `'center'` o `'edge'`. El valor `'center'` indica que la barra se centrará en la posición indicada, mientras que el valor `'edge'` indica que el borde izquierdo de la barra se alineará con el valor de la x.
- `color`, que indica el color de las barras.
- `tick_label`, una cadena o lista de cadenas que indica los nombres dados a las barras en el eje horizontal.
- `orientation`, que puede tomar los valores `'vertical'` y `'horizontal'` e indica si las barras se colocan vertical u horizontalmente, respectivamente. Su valor por defecto es `'vertical'`.
- `yerror`, que es un escalar, una lista de escalares o una lista de parejas de escalares. Cuando para cada barra corresponde un único escalar se mostrará una línea de error en el extremo superior simétrica para el error positivo y negativo. Si corresponden dos escalares el primero corresponderá al posible exceso y el segundo al posible defecto.
- `xerror`, que funciona como `yerror` pero se aplica en horizontal.

Esta función devuelve un contenedor con los atributos gráficos de las barras y las líneas de error. Para ilustrar el uso básico de estos diagramas empezamos mostrando los valores absolutos de paro para una serie de ciudades dadas como parámetro.

Para ello usamos la función auxiliar `datos_ciudades` que, dada la ruta del fichero y la lista de ciudades de interés, devuelve la lista de ciudades reordenada según el orden en el que ha encontrado las ciudades en el fichero y las cantidades totales de paro. Así, tendríamos los siguientes valores para una llamada para Madrid, Barcelona, Valencia y Teruel:

```
>>> datos_ciudades(ruta_datos, ["COM. DE MADRID", "BARCELONA",
"VALENCIA", "TERUEL"])
(['TERUEL', 'BARCELONA', 'VALENCIA', 'COM. DE MADRID'],  [5922.0,
300790.0, 188918.0, 380051.0])
```

La función `barras_ciudades`, a cargo de dibujar la gráfica, llama a la función auxiliar y almacena los resultados en variables, crea índices para situar las barras y define un ancho para ellas. Crea entonces unos nuevos ejes y usa la función `bar` para crear el diagrama con barras rojas y los datos especificados. Por último, usamos funciones vistas anteriormente para darle un nombre al eje vertical, ajustar el *layout* y mostrar la gráfica.

```
def barras_ciudades(ruta, ciudades):
    etqs, datos = datos_ciudades(ruta, ciudades)

    ind = np.arange(4)  # índices para las X
    ancho = 0.35        # Ancho de las barras

    fig, eje = plt.subplots()
    barras = eje.bar(ind, datos, ancho, color='r', tick_label=etqs)
    plt.ylabel('Cantidad')
    plt.tight_layout()
    plt.show()
```

La siguiente llamada:

```
>>> ciudades = ["TERUEL", "BARCELONA", "VALENCIA", "COM. DE MADRID"]
>>> barras_ciudades(ruta_datos,ciudades)
```

genera la gráfica que vemos en la figura 8-4 para las ciudades de Teruel, Barcelona, Valencia y Madrid. Aunque, como hemos visto en los apartados anteriores, los porcentajes de paro por tramos de edad son parecidos los datos absolutos varían mucho dependiendo de la población de la ciudad. Así, podemos comprobar que el paro absoluto para Teruel es mucho menor que para el resto de ciudades consideradas, al tratarse de una ciudad de población mucho menor.

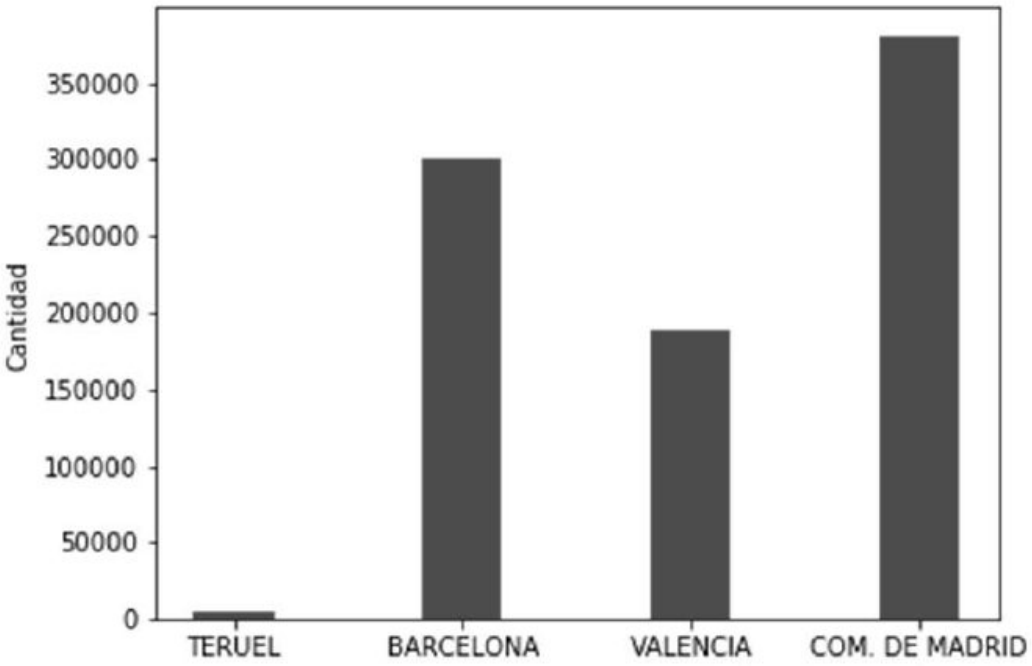


Figura 8-4. Valores absolutos de paro para Teruel, Barcelona, Valencia y Madrid.

Un uso habitual del diagrama de barras es estudiar diferentes poblaciones dentro de una misma categoría. Por ejemplo, podemos estudiar los valores absolutos de paro por tramos de edad, separando los valores de hombres y mujeres. Para ello usaremos la función auxiliar `datos_provincia_totales`, que dada la ruta del fichero devuelve una pareja de listas con el total, por tramos de edad, de paro de hombres y mujeres, respectivamente. Así, la función `totales_barras`, que presentamos a continuación, primero llama a esta función auxiliar para tener los valores disponibles, luego crea unos índices para situar las barras y define una variable `ancho` para la anchura de las barras. Creamos entonces unos nuevos ejes y una lista de etiquetas, que usaremos en lugar de las indicadas en el fichero para asegurarnos de que caben en el eje. En primer lugar, creamos las barras para los hombres, que serán en color azul. Para colocar ahora las barras de las mujeres junto a las anteriores lo que haremos es desplazar nuestros índices el ancho especificado para las barras, con lo que forzaremos que ambas estén juntas. Además, indicaremos que las barras de las mujeres estén en color amarillo y colocaremos las etiquetas a su altura. Por último, daremos nombre al eje vertical y colocaremos la leyenda.

```
def totales_barras(ruta):
    hom, muj = datos_provincia_totales(ruta)

    ind = np.arange(4)  # índices para las X
    ancho = 0.35        # Ancho de las barras
    fig, eje = plt.subplots()
```

```
    etqs = ["Menores 25", "25-29", "30-44", "Mayores 45"]
    barrasH = eje.bar(ind, hom, ancho, color='b')
    barrasM = eje.bar(ind + ancho, muj, ancho, color='y',
                      tick_label=etqs)

    plt.ylabel('Cantidad')
    plt.legend((barrasH[0], barrasM[0]), ('Hombres', 'Mujeres'))
    plt.tight_layout()
    plt.show()
```

La llamada necesaria para ejecutar este código es

```
>>> totales_barras(ruta_datos)
```

Esta llamada crea el diagrama mostrado en la figura 8-5, donde vemos que el paro es en general mayor para las mujeres, excepto en la franja de menores de 25 años, en la que el paro es mínimamente superior en hombres.

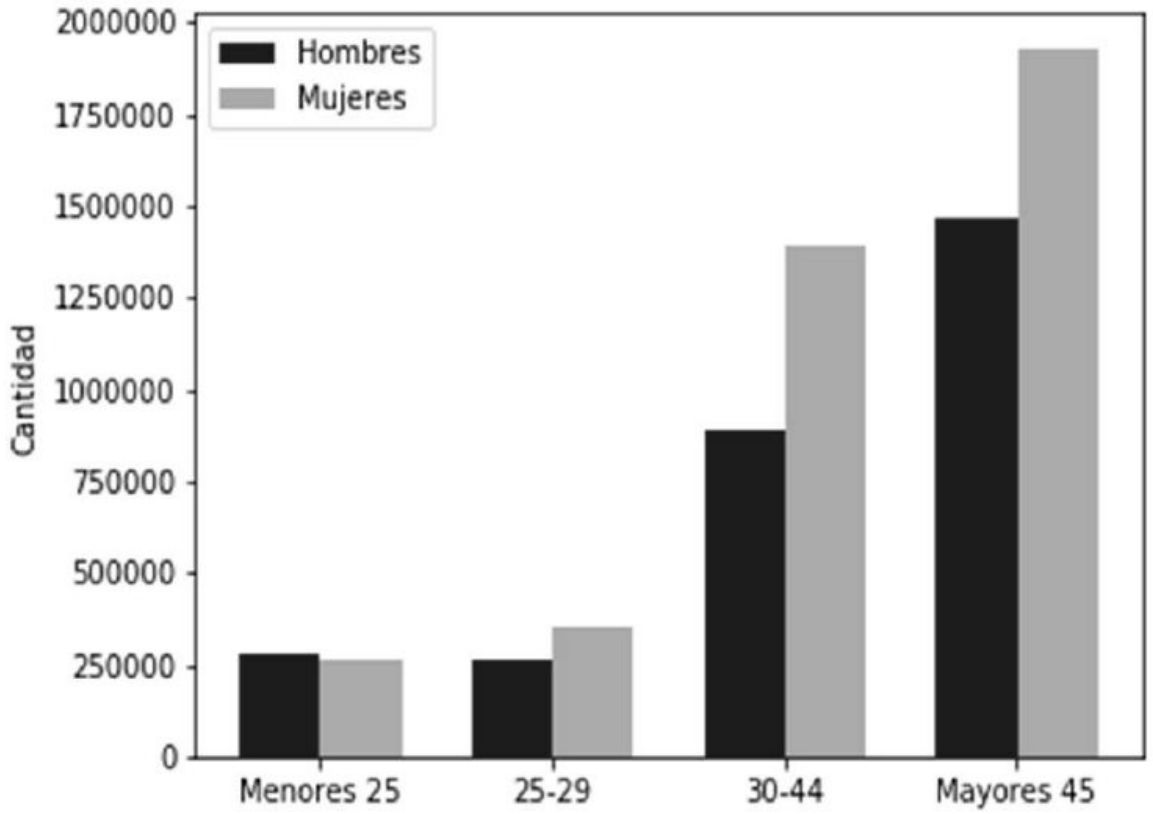


Figura 8-5. Totales de paro por tramos de edad, distinguiendo hombres y mujeres.

El diagrama anterior nos permite fácilmente comparar entre hombres y mujeres, pero dificulta el cálculo de los totales por categoría. Este cálculo se simplifica si las

barras se encuentran una sobre otra, comportamiento que implementa la función `totales_barras_pila`. Esta función sigue el mismo esquema que el mostrado arriba, pero al generar la barra para las mujeres usamos el argumento `bottom` para que la barra no empiece a altura 0, sino en la posición en la que termina la barra para los hombres:

```
def totales_barras_pila(ruta):
    hom, muj = datos_provincia_totales(ruta)

    ind = np.arange(4)  # índices para las X
    ancho = 0.35        # Ancho de las barras

    fig, eje = plt.subplots()
    etqs = [“Menores 25”, “25-29”, “30-44”, “Mayores 45”]
    barrasH = eje.bar(ind, hom, ancho, color=’b’)
    barrasM = eje.bar(ind, muj, ancho, bottom=hom, color=’y’,
                      tick_label=etqs)

    plt.ylabel(‘Cantidad’)
    plt.legend((barrasH[0], barrasM[0]), (‘Hombres’, ‘Mujeres’))

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

Usaremos esta función con la llamada:

```
>>> totales_barras_pilas(ruta_datos)
```

Esta llamada genera la figura mostrada en la figura 8-6, en la que fácilmente vemos los totales de paro por tramos de edad a la vez que podemos comparar el paro entre hombres y mujeres.

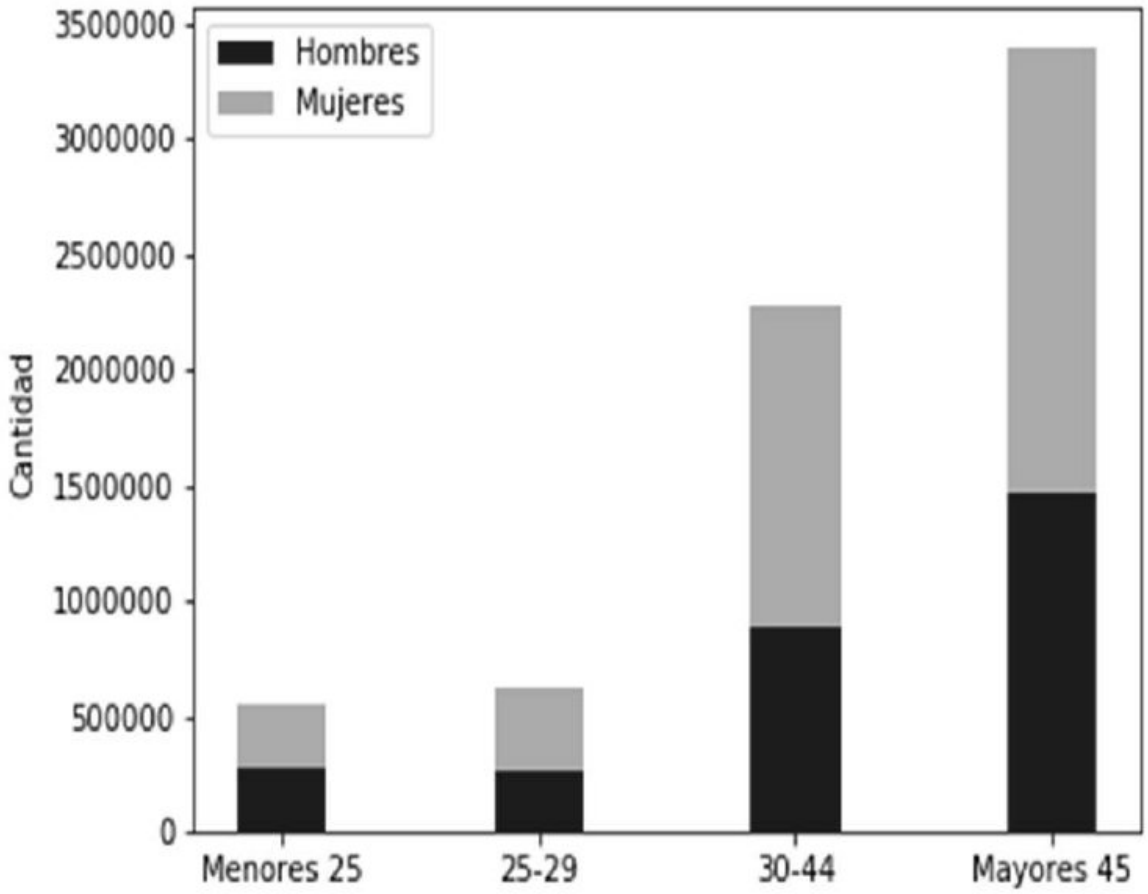


Figura 8-6. Totales de paro apilados por tramos de edad, distinguiendo hombres y mujeres.

Histograma

Un histograma es una gráfica de barras en la que, a diferencia de lo que veíamos en el apartado anterior, la altura de cada barra indica la frecuencia de aparición de los valores en el tramo dado. La biblioteca `matplotlib` proporciona la función `hist` para generar histogramas. Esta función admite los siguientes argumentos:

- `x`, una lista o lista de listas de valores para los que queremos crear el histograma. Si se da una lista de valores no existe la restricción de que todas las listas tengan la misma longitud. Este argumento es el único obligatorio.
- `bins`, un entero, una secuencia o `auto`. Este parámetro indica el número de contendores (o barras) que se utilizarán en el histograma. Si se usa un número `n` entero entonces Python usará `n + 1` contenedores. Si se da una secuencia [a, b, c, ..., z] entonces tendremos tramos en los que el extremo

izquierdo está incluido pero el derecho excluido, excepto para el último tramo, en el que se incluyen ambos. Así, la secuencia [1,2,3,4] indica que el primer tramo es [1, 2), el segundo [2, 3) y el último [3,4]. Por último, si tenemos instalado NumPy 1.11 o superior podemos usar la opción `auto`, que trata de optimizar la presentación.

- `range`, una pareja de enteros que indica el rango de valores a mostrar y que se inicializa por defecto al mínimo y al máximo de la muestra. El parámetro `range` no tiene efecto si el parámetro `bins` es una secuencia.
- `cumulative`, un booleano que por defecto toma el valor `False`. Cuando toma el valor `True` indica que cada barra acumulará la cantidad de valores de todas las anteriores.
- `bottom`, un escalar o una lista de escalares que indica la altura en la que empiezan las barras. Si su valor es un escalar todas las barras empezarán en esa altura, mientras que en el caso de tener una lista cada barra empezará en la altura correspondiente. Por defecto su valor es `None` e indica que empiezan a altura 0.
- `orientation`, que puede tomar los valores `'horizontal'` o `'vertical'` (por defecto tiene el valor `'vertical'`) e indica la orientación de las barras.
- `color`, un *string* o una lista de *string* de colores, que indica los colores de las barras en el histograma.

La función devuelve una tupla con:

- `n`, una lista o lista de listas con los valores en las barras.
- `bins`, un array con las aristas de las barras.
- `patches`, una lista o lista de listas con los elementos gráficos de las barras.

Para ilustrar cómo usar esta función definiremos una función que muestra en un histograma los porcentajes de paro de todas las ciudades para un cierto tramo de edad, que definiremos como un entero que toma los valores 0 (menores de 25), 1 (entre 25 y 29), 2 (entre 30 y 44) y 3 (mayores de 45). Para ello usamos dos funciones auxiliares:

- `porcentajes_edad`, función que devuelve una lista con todos los porcentajes, dada la ruta y el tramo a analizar.
- `string_tramo`, que crea un *string* para usar en el título indicando el tramo de edad seleccionado.

En particular, la primera función aplicada al primer tramo devuelve una lista de 53 elementos:

```
>>> porcentajes_edad(ruta_datos, 0)
[8.286365271615301,..., 8.016476598887788]
```

La función principal, `histograma_tramo`, recibe la ruta del fichero y el tramo de interés y, en primer lugar, usa la función `porcentajes_edad` para obtener la lista con los porcentajes. Entonces creamos unos nuevos ejes y usamos la función `hist` para generar el histograma. Ahora usamos la función auxiliar `string_tramo` para crear el título y damos nombre a los ejes con las funciones presentadas al principio del capítulo:

```
def histograma_tramo(ruta, tramo):
    l = porcentajes_edad(ruta, tramo)

    plt.subplots()
    plt.hist(l)

    msj = string_tramo(tramo)
    plt.xlabel('Porcentaje de parados')
    plt.ylabel('Porcentaje de ciudades con ese paro')
    plt.title('Porcentajes de paro en el tramo ' + msj)
    plt.show()
```

La llamada que usaremos para calcular el histograma es:

```
>>> histograma_tramo(ruta_datos, 3)
```

La figura 8-7 muestra el resultado de esta llamada. Como vimos en el diagrama de caja, la mayoría de los valores están alrededor del 50%, aunque también hay bastantes valores cercanos al 55%.

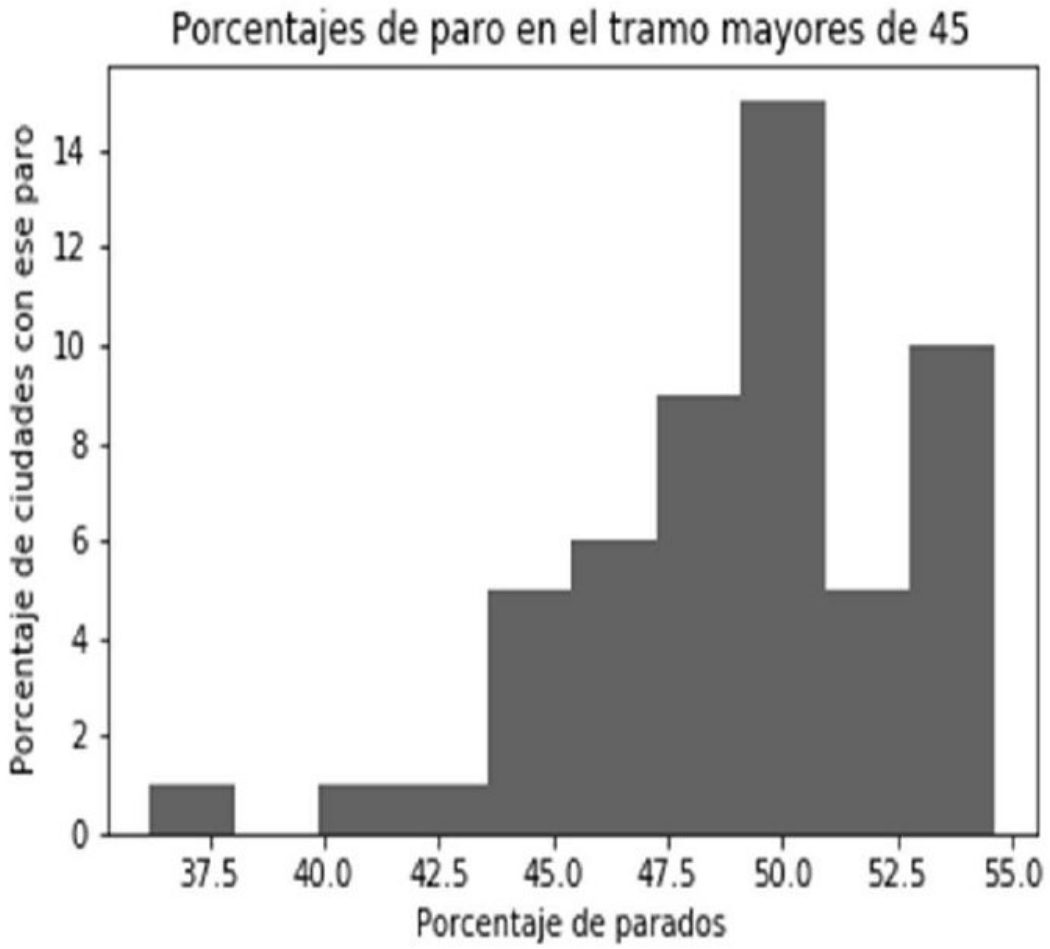


Figura 8-7. Histograma con el porcentaje de parados en el tramo de mayores de 45 años.

CONCLUSIONES

En este capítulo hemos visto formas sencillas pero eficaces de visualizar la información en Python. Este capítulo concluye el proceso que hemos seguido en este libro, donde hemos aprendido a almacenar los datos, a leerlos y a procesarlos para ser capaces de obtener información.

Esperamos que todo este proceso permita al lector dar un salto de calidad a la hora de entender la información "a su alrededor" en su beneficio, permitiendo la toma de decisiones informadas.

REFERENCIAS

- Allen Yu, Claire Chung y Aldrin Yim. Matplotlib 2.x by example. Packt Publishing, 2017.
- Matt A. Wood. Python and Matplotlib essentials for scientists and engineers. IOP Concise Physics, 2015.
- Duncan M. McGreggor. Mastering Matplotlib. Packt Publishing, 2015.
- Documentación de la función `plot` (accedida en abril de 2018). https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot.
- Documentación sobre formateo de strings (accedido en abril de 2018). https://docs.python.org/3/library/string.html

A INSTALACIÓN DEL SOFTWARE

INTRODUCCIÓN

En este apéndice mostraremos cómo instalar todo el software utilizado a lo largo del libro, concretamente Python y sus bibliotecas, MongoDB y Apache Spark. Daremos instrucciones para Windows 10, Linux y MacOS X versión 10.13. En lo relativo a Linux supondremos una distribución Ubuntu, aunque las instrucciones serán equivalentes para otros sistemas.

PYTHON Y SUS BIBLIOTECAS

Python es un lenguaje de programación interpretado disponible en Windows, Linux y Mac. Actualmente conviven las versiones 2.7 y la 3, aunque la versión 2.7 dejará de recibir actualizaciones en 2020. Por ello en este libro hemos utilizado la versión más reciente de la rama de Python 3, la 3.6. En algunos sistemas como Linux, Python aparece instalado por defecto, aunque por comodidad y uniformidad vamos a tratar la instalación de Python con Anaconda en todos los sistemas. Anaconda es una distribución Python muy popular que aglutina varios paquetes para análisis de datos y aprendizaje automático, por lo que simplifica bastante la instalación en cualquier sistema.

Windows 10

La instalación de Anaconda es muy sencilla; basta con acceder a la página de Anaconda y descargar el ejecutable correspondiente:

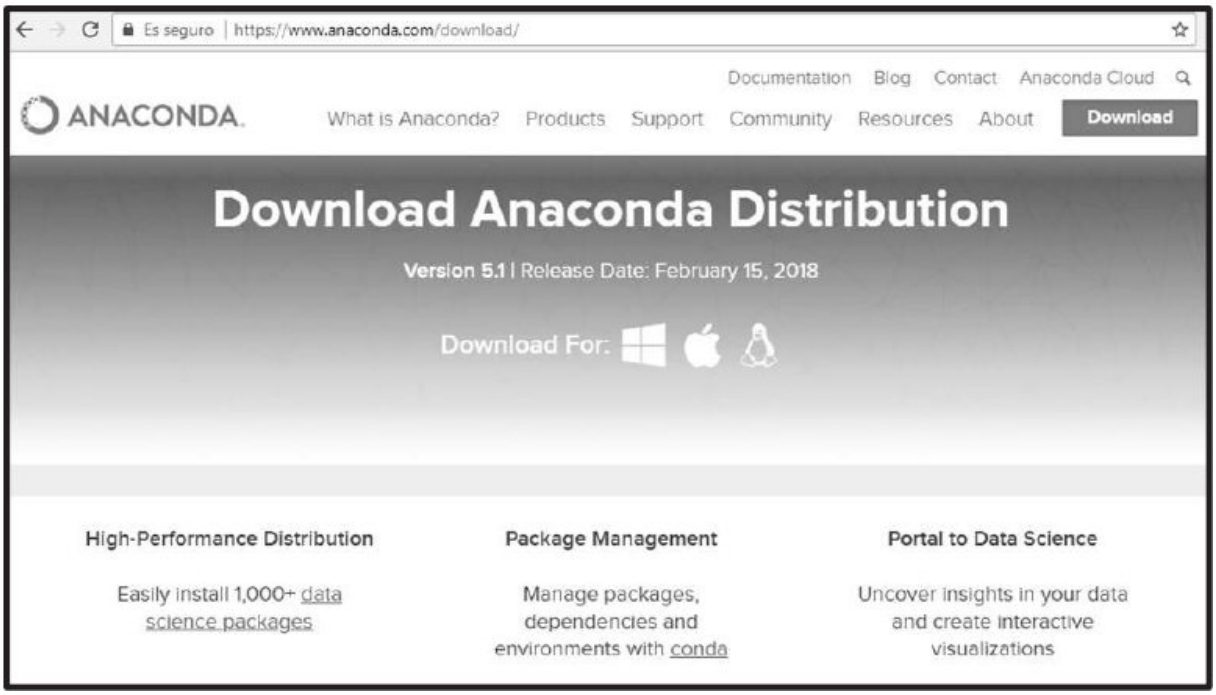


Figura A-1. Página de descarga de Anaconda.

A partir de ahí, simplemente instalamos el software y ya tendremos listo Anaconda. En el grupo creado en el menú de inicio para Anaconda podremos ver un componente con el nombre Jupyter Notebook. Hacemos clic sobre él y se abrirá un terminal, que tras ciertos mensajes de inicialización indicará algo así:

```
Copy/paste this URL into your browser when you connect for the
first time, to login with a token:

http://localhost:8888/?token=d2186d083e73d50bffe9929f94e617060a475ba4
ece60491
```

Basta con copiar esta URL en nuestro navegador para tener acceder a la página principal de los Jupyter Notebooks. Normalmente ni siquiera esto es necesario, porque el propio programa de inicialización habrá abierto el navegador. En todo caso, la página tendrá este aspecto:

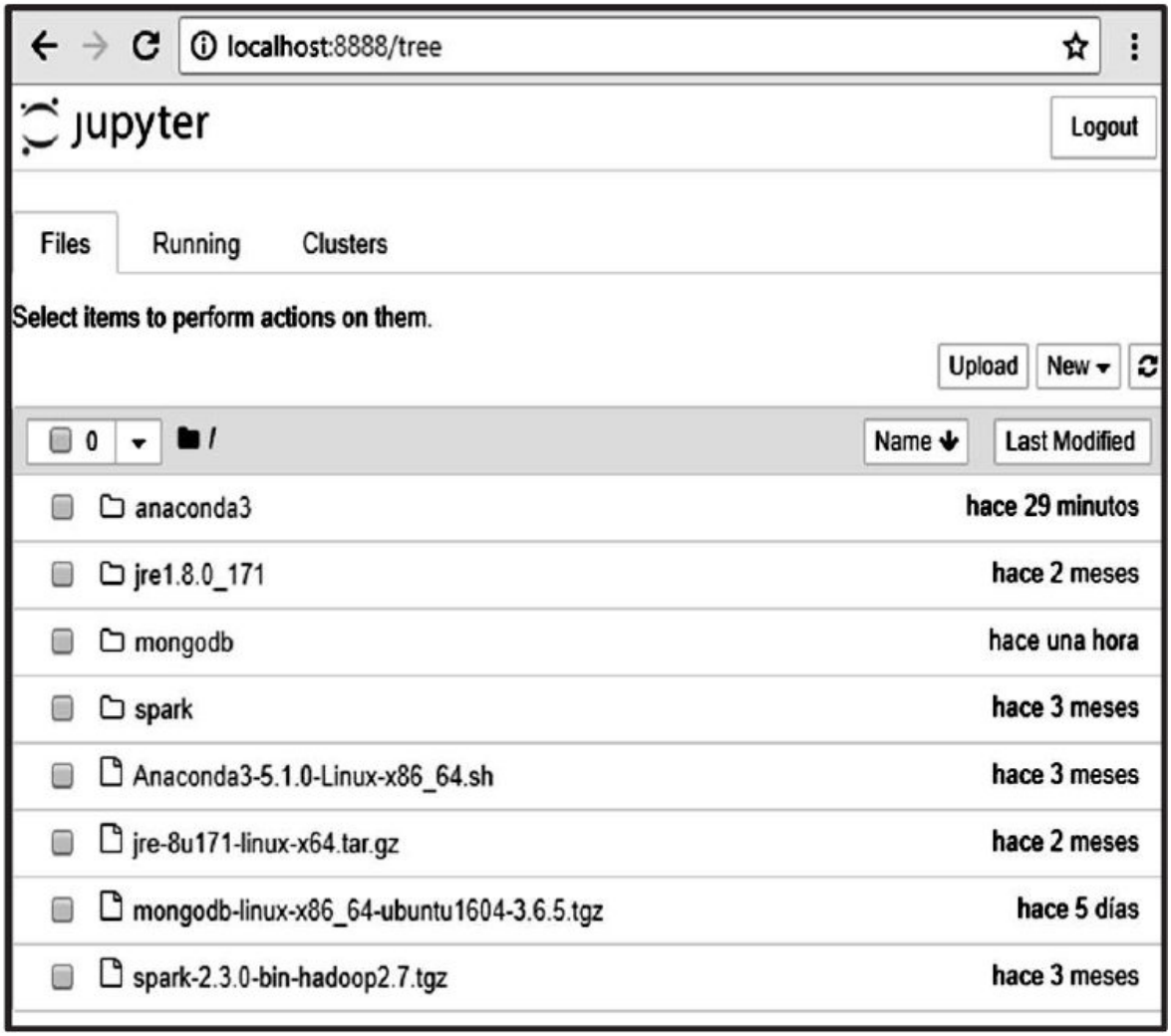


Figura A-2. Página principal del entorno Jupyter.

A partir de aquí podremos manejar nuestros notebooks. Si ya tenemos uno creado y deseamos continuar con él, podemos hacerlo eligiendo de la lista mostrada por esta página. En caso de querer acceder a un notebook que hayamos descargado, por ejemplo desde una página web, emplearemos *Upload* (arriba a la derecha, ver Figura A-2). Si por el contrario deseamos crear uno nuevo, podremos emplear el botón de al lado de upload, *New*, y seleccionar Python 3.

Las instrucciones para cerrar los notebook, así como para instalación de bibliotecas nuevas, coinciden con las explicadas en la descripción de la instalación de Linux, a continuación.

Linux

Para instalar Anaconda en un sistema Linux, deberemos acceder a `https://www.anaconda.com/download/#linux` (ver figura A-1) y seleccionar el paquete de la versión 3.6 que más se ajuste a nuestro sistema (normalmente la de 64 bits para plataforma *x86*). Esto descargará un fichero de extensión `.sh` de aproximadamente 500 MB. Este fichero se deberá lanzar desde un terminal con el siguiente comando (suponiendo que el fichero descargado tiene nombre `Anaconda3-5.1.0-Linux-x86_64.sh`):

```
$ sh Anaconda3-5.1.0-Linux-x86_64.sh
```

Esto lanzará un proceso de instalación que puede requerir varios minutos. En algún momento nos preguntará si deseamos insertar la ruta de Anaconda a nuestra variable de entorno `PATH`, a lo que contestaremos que sí, puesto que esto facilitará la instalación del resto de programas:

```
Do you wish the installer to prepend the Anaconda3 install location
to PATH in your /home/<USER>/.bashrc ? [yes|no]
[no] >>> yes
Appending source /home/<USER> /anaconda3/bin/activate to
/home/<USER>/.bashrc
A backup will be made to: /home/<USER>/.bashrc-anaconda3.bak
```

Una vez finalizado este proceso abrimos una nueva terminal (para recargar la variable de entorno `PATH`) y comprobamos que Python se ha instalado correctamente. Debe mostrarse la versión de Python y el indicador de que se trata de la distribución Anaconda:

```
$ python
Python 3.6.5 |Anaconda, Inc. | (default, Apr 29 2018, 16:14:56)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
```

Para lanzar Jupyter, el entorno Python utilizado en este libro, será necesario invocar al comando:

```
$ jupyter notebook
```

Se mostrarán algunos mensajes en el terminal y automáticamente se abrirá el navegador predeterminado del sistema mostrando la pantalla principal de Jupyter, como en la figura A-2. Si el arranque automático del navegador fallase, o si se

prefiere utilizar otro navegador diferente del navegador por defecto, se puede consultar el terminal buscando un mensaje similar a este:

```
Copy/paste this URL into your browser when you connect for the first
time,
    to login with a token:
        http://localhost:8888/?token=580b3ae727b5b9d0c2...04d88aff17b
```

Si se accede a dicha URL desde cualquier navegador del sistema también se abrirá la página principal de Jupyter. Para cerrar Jupyter se debe primero finalizar y guardar todos los *notebooks* abiertos y finalmente pulsar la combinación de teclas *Ctrl-C* en el terminal dese el que se lanzó Jupyter. Se pedirá confirmación para cerrar Jupyter, a lo que se deberá contestar *yes*:

```
Shutdown this notebook server (y/[n])? y
[C 11:17:30.110 NotebookApp] Shutdown confirmed
[I 11:17:30.112 NotebookApp] Shutting down 0 kernels
```

La instalación de bibliotecas Python en Linux se realizará mediante el comando `pip`, que habrá sido instalado por Anaconda y estará en el `PATH`. `pip` es un gestor de paquetes para Python, y es la manera recomendada para instalar, eliminar o actualizar bibliotecas Python. Para instalar un paquete invocaremos a `pip install` desde el terminal y a continuación escribiremos el nombre del paquete. Por ejemplo, para instalar la biblioteca `pymongo` de conexión entre Python y MongoDB ejecutaremos:

```
$ pip install pymongo
```

En cualquier momento podremos comprobar las bibliotecas instaladas y sus versiones mediante `pip list`. También podremos actualizar bibliotecas previamente instaladas a su versión más reciente mediante:

```
$ pip install --upgrade <nombre_biblioteca>
```

Mac OS

La instalación de Python en Mac OS es muy similar a la explicada para Linux. En primer lugar, accederemos a la página (análoga a la mostrada en la figura A-1):

```
https://www.anaconda.com/download/#macos
```

y descargaremos el fichero .pkg correspondiente a Python 3.6. Una vez descargado, haremos doble clic en el fichero y accederemos al instalador automático, donde aceptaremos todas las opciones por defecto. Este proceso añadirá la aplicación "Anaconda-Navigator" en la carpeta Aplicaciones, con lo que es suficiente hacer doble clic en la aplicación para ejecutarla y obtener una ventana similar a la mostrada en la figura A-3. En esta ventana basta pulsar el botón "Launch" de Jupyter (ventana superior izquierda) para lanzar la aplicación y llegar al navegador de ficheros en la figura A-2. Para ello, el sistema lanzará automáticamente un terminal con el comando mostrado en el apartado anterior, por lo que podemos seguir las mismas instrucciones para cerrarlo.

De la misma manera, nuestra instalación de Anaconda habrá instalado `pip` en el sistema, por lo que podemos abrir un terminal (disponible en la carpeta Aplicaciones/Utilidades) y seguir las mismas instrucciones mostradas anteriormente.

Figura A-3. Anaconda Navigator en Mac OS.

MONGODB

La versión de MongoDB que vamos a utilizar en este libro es la *Comunity Server* 4.0, que está disponible bajo licencia abierta *GNU Affero*.

Windows 10

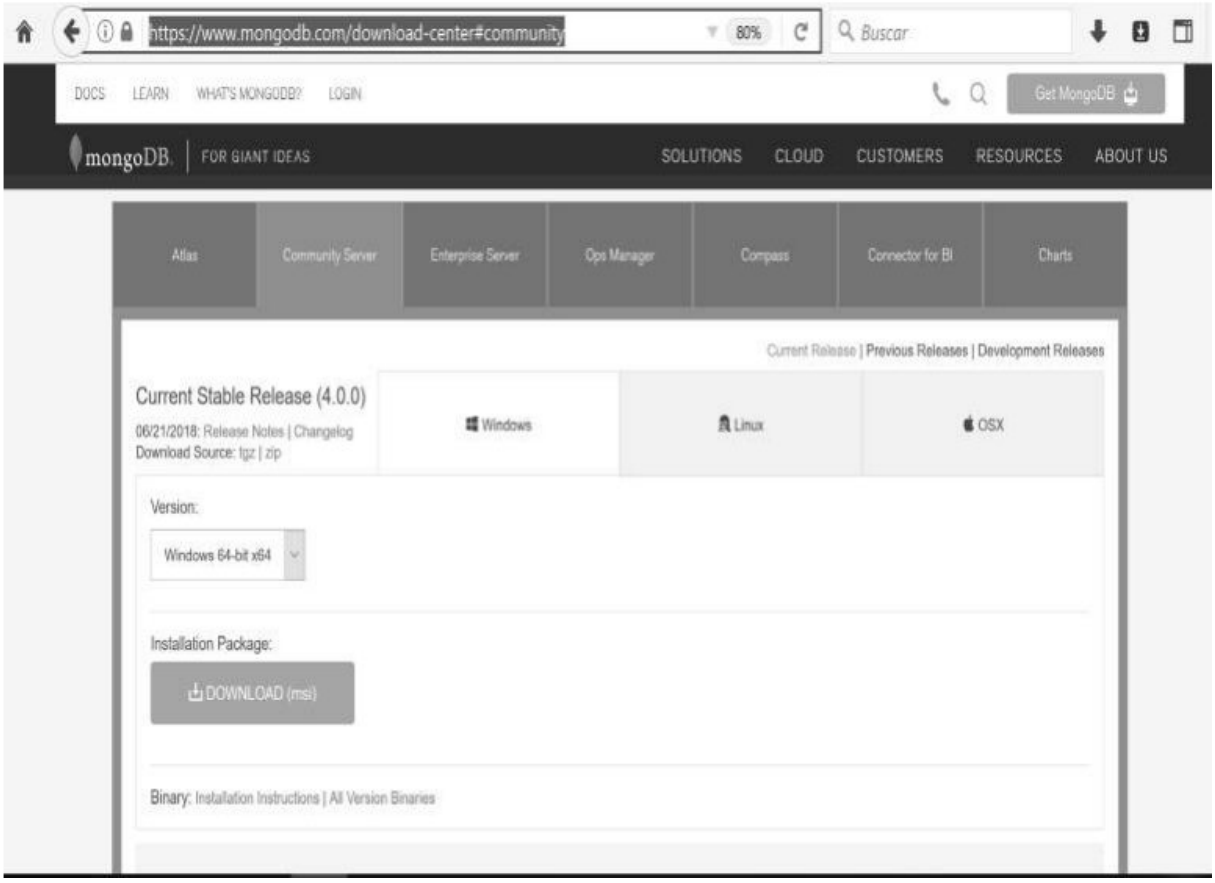


Figura A-4. Página de descarga de Mongo Community Server.

Allí podemos descargar e instalar el fichero de instalación (msi). Conviene indicar que se desea instalar la versión completa, prácticamente todos los componentes son interesantes y no ocupan mucho.

En un momento dado, Mongo nos preguntará si queremos instalar la base de datos como un servicio, tal y como indica la figura A-5:

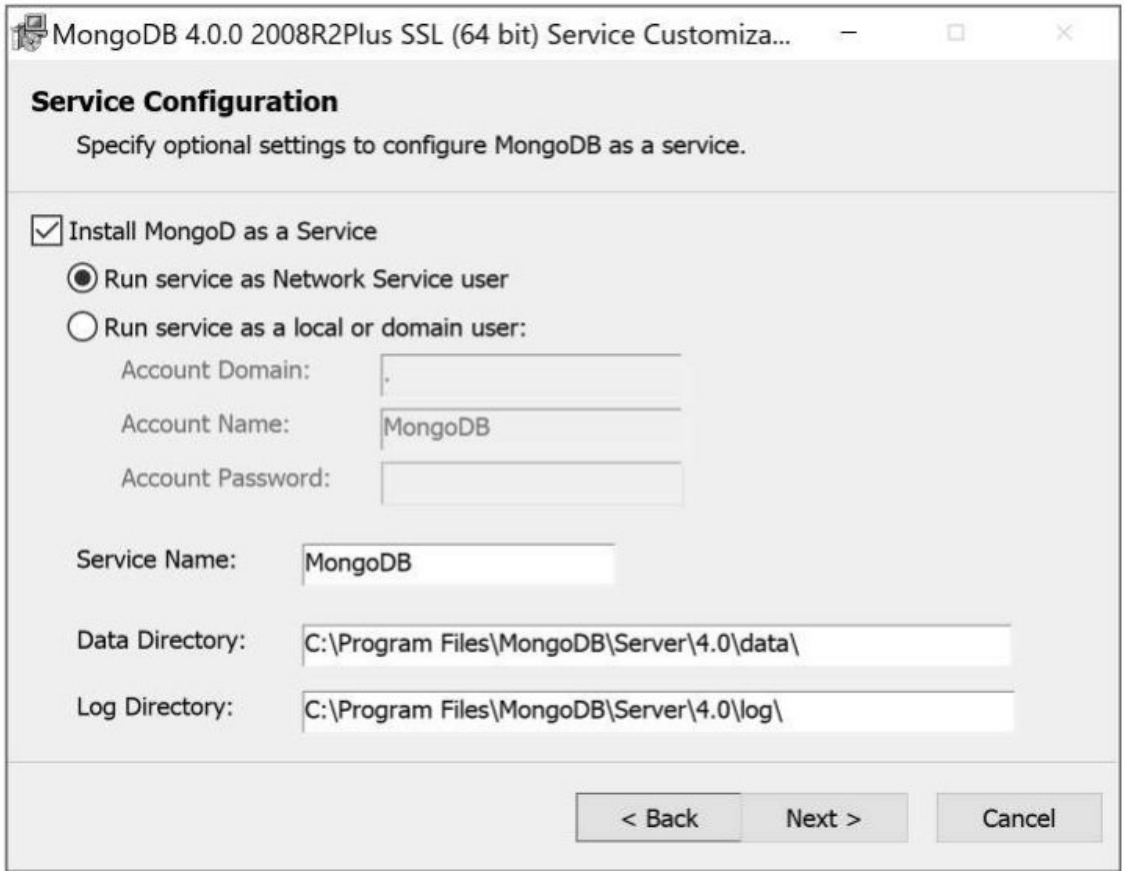


Figura A-5. Posibilidad de instalar Mongo como servicio.

Si decimos que sí (opción marcada por defecto), el servidor de Mongo se iniciará de forma automática cada vez que arranque Windows. Esto tiene la ventaja de que no nos tendremos que encargar de arrancar el servidor manualmente. La desventaja, a cambio, será que:

1. Retrasará el inicio de Windows, incluso si no vamos a usar MongoDB.
2. A menudo es preferible iniciar el servidor, o varios servidores en directorios de datos distintos, para tener instancias diferentes en diversos lugares. Esto puede convenir por razones de seguridad, para facilitar backups, etc.
3. Desde un punto de vista de aprendizaje es buena idea "obligarse" a recordar que la arquitectura cliente-servidor obliga a iniciar el servidor previamente.

En este libro asumimos que no se elige la opción de instalar Mongo como servicio, es decir, que se desmarca la casilla correspondiente antes de hacer clic en el botón "Next". Al finalizar, y para poder iniciar tanto mongod (servidor) como

mongo (cliente) y utilizar las herramientas mongoimport, etc., conviene añadir a la de entorno PATH la ruta al directorio bin de la instalación de Mongo, por ejemplo.

Para ello desde el explorador de ficheros de Windows, nos situamos sobre el icono *Este Equipo* y pulsamos el botón derecho, seleccionando *Propiedades*. En la ventana que se nos abre hacemos clic sobre *Opciones avanzadas*. Se abrirá una nueva ventana similar a la de la figura A-6.

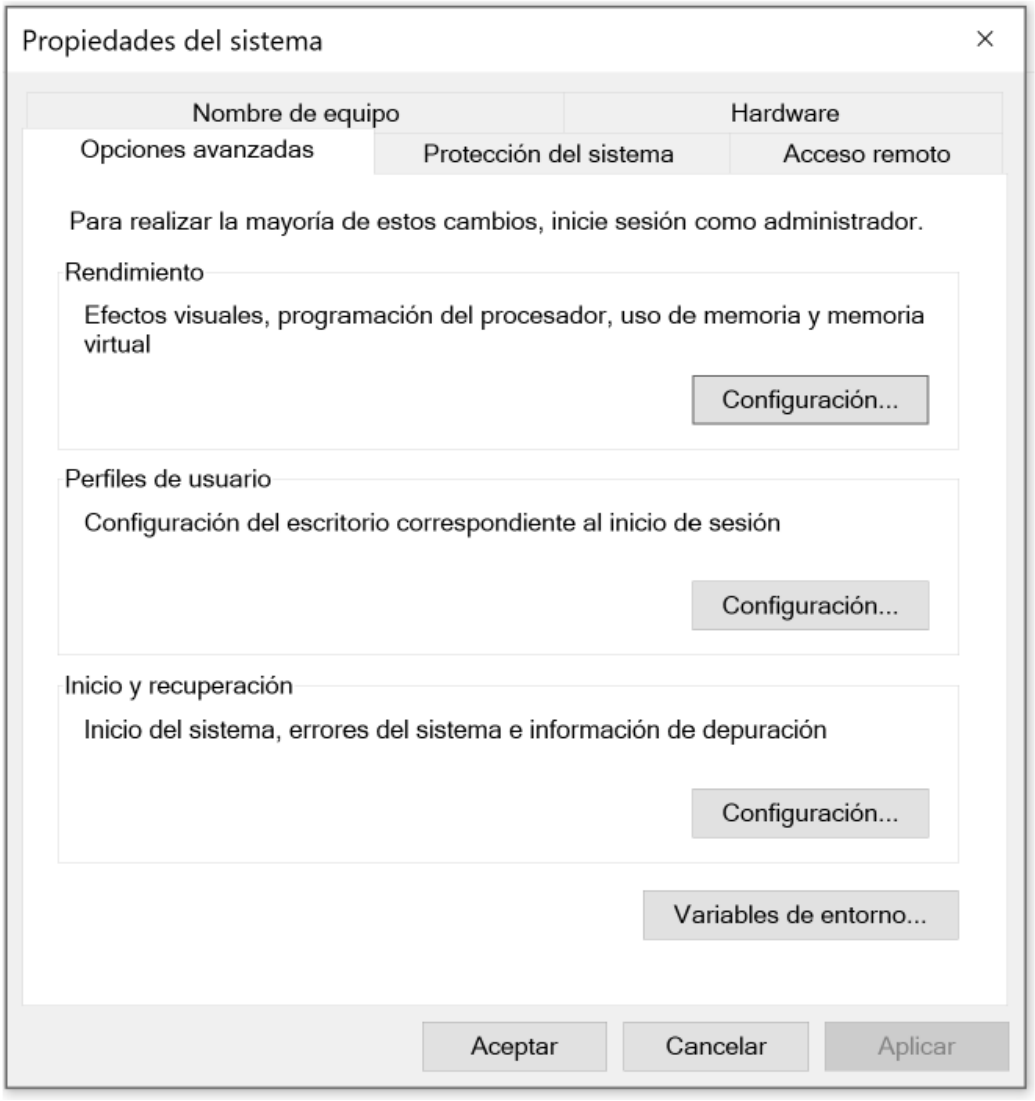


Figura A-6. Ventana de Propiedades del Sistema.

En esta ventana seleccionamos el botón *Variables de Entorno*, y de la lista de variables del sistema, seleccionamos la variable Path, pulsando a continuación *Editar*. En la ventana que se abre, y que tiene todos los directorios del Path, podemos pulsar *Nuevo* y añadir la carpeta, por ejemplo *C:\Program Files\MongoDB\Server\4.0\bin*. Tras pulsar *Aceptar* y cerrar todas las ventanas, podremos abrir un *Símbolo del*

sistema y tanto los comandos `mongo` como `mongod` o `mongoimport` deben estar accesibles.

Linux

Para instalar MongoDB en un sistema Linux lo primero que debemos hacer es acudir a la página de descargas de la versión *Community Server* disponible en `https://www.mongodb.com/download-center?jmp=nav#community` (ver figura A-4). En esta página seleccionaremos *Linux* y elegiremos en el menú desplegable la versión adecuada para nuestro sistema. MongoDB puede ser instalada desde los gestores de paquetes `yum` (para RedHat y derivados) o `apt` (para Debian y derivados), y la propia página de descargas incluye información para realizar la instalación de este modo. Sin embargo, en esta sección mostraremos cómo instalarlo a partir del fichero `.tgz`, puesto que funcionará para todas las versiones de Linux.

Lo primero que deberemos hacer tras elegir la versión adecuada para nuestra distribución Linux es pulsar el botón "*Download (tgz)*" y descargar el fichero en nuestro directorio personal. Si queremos instalar MongoDB en otro directorio solo tendremos que mover este fichero antes de continuar con el resto de instrucciones. A continuación, abriremos un terminal, nos situaremos en la carpeta donde hemos situado el fichero `.tgz` y lo descomprimiremos con:

```
$ tar xvaf mongodb-linux-x86_64-ubuntu1604-xxxx.tgz
```

Esto creará una nueva carpeta llamada `mongodb-linux-x86_64-ubuntu1604xxxx` que contendrá un directorio `bin` con los binarios de MongoDB. Por simplicidad, renombraremos esta carpeta a `mongodb` puesto que nos permitirá actualizar la versión de MongoDB simplemente reemplazando esta carpeta.

```
$ mv mongodb-linux-x86_64-ubuntu1604-xxxx mongodb
```

Para facilitar la invocación del servidor y el cliente añadiremos este directorio a la variable de entorno `PATH` de nuestro sistema editando el fichero `~/.bashrc` e incluyendo al final:

```
# Incluimos MongoDB en el PATH
export PATH=/home/<user>/mongodb/bin:$PATH
```

Abrimos un nuevo terminal para recargar el `PATH` y ya podremos invocar al servidor y al cliente de MongoDB. Para lanzar el servidor ejecutaremos:

```
$ mongod --dbpath <carpeta_datos>
```


Es importante utilizar el parámetro `--dbpath` para indicar la ruta donde queremos que MongoDB almacene los datos de las bases de datos, puesto que de otra manera tratará de acceder a la carpeta `/data`. Como esta carpeta no existe de manera predeterminada en las distribuciones Linux, invocar a `mongod` sin ningún parámetro producirá que el servidor se aborte de manera prematura con el error:

```
NonExistentPath: Data directory /data/db not found., terminating
```

Para lanzar el cliente de MongoDB para conectarse al servidor local usando el puerto por defecto deberemos invocar:

```
$ mongo
MongoDB shell version vxxxx
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: xxxx
(…)
>
```

Si hubiésemos olvidado arrancar previamente el servidor, o si no pudiese conectarse con él, el cliente de MongoDB nos mostraría un error de conexión:

```
$ mongo
MongoDB shell version vxxx
connecting to: mongodb://127.0.0.1:27017
(…)
exception: connect failed
```

Se pueden var más detalles sobre la arquitectura cliente-servidor de Mongo en el capítulo dedicado a esta base de datos.

Mac OS

Es posible instalar Mongo en Mac OS siguiendo los mismos pasos que hemos visto arriba para Linux. Sin embargo, si tenemos instalado Homebrew es posible usarlo para realizar una instalación más sencilla (recuerda ejecutar `brew update` en el terminal para tener la versión más actual). Simplemente ejecutaremos el comando:

```
$ brew install mongodb
```

Podemos crear ahora el directorio `/data/db` y darle los permisos adecuados:

```
$ mkdir -p /data/db
$ sudo chown -$ `id un` /data/db
```

Podemos también en este punto incluir el directorio en el `PATH` para facilitar futuros usos, como se explicó anteriormente. Uno vez ejecutados estos pasos ya estamos listos para ejecutar el servidor con el comando `mongod` y y MongoDB con el comando `mongo`.

APACHE SPARK Y PYSPARK

La instalación de pyspark en el entorno Windows ha sido siempre una tarea ardua. Sin embargo, en las últimas versiones es tan sencillo como añadir una librería nueva en Python.

Windows 10

Los requisitos previos son disponer de Anaconda (cuya instalación se describe en este mismo apéndice) y disponer de una versión actualizada de Java JRE.

A partir de aquí bastará con teclear, desde cualquier símbolo del sistema:

```
pip install pyspark
```

Nota: es conveniente abrir el símbolo del sistema como administrador. Para ello desde el menú de inicio tecleamos *Símbolo del Sistema*, y a continuación pulsamos el botón derecho sobre la aplicación eligiendo *Ejecutar* como administrador, tal y como muestra la figura A-7.

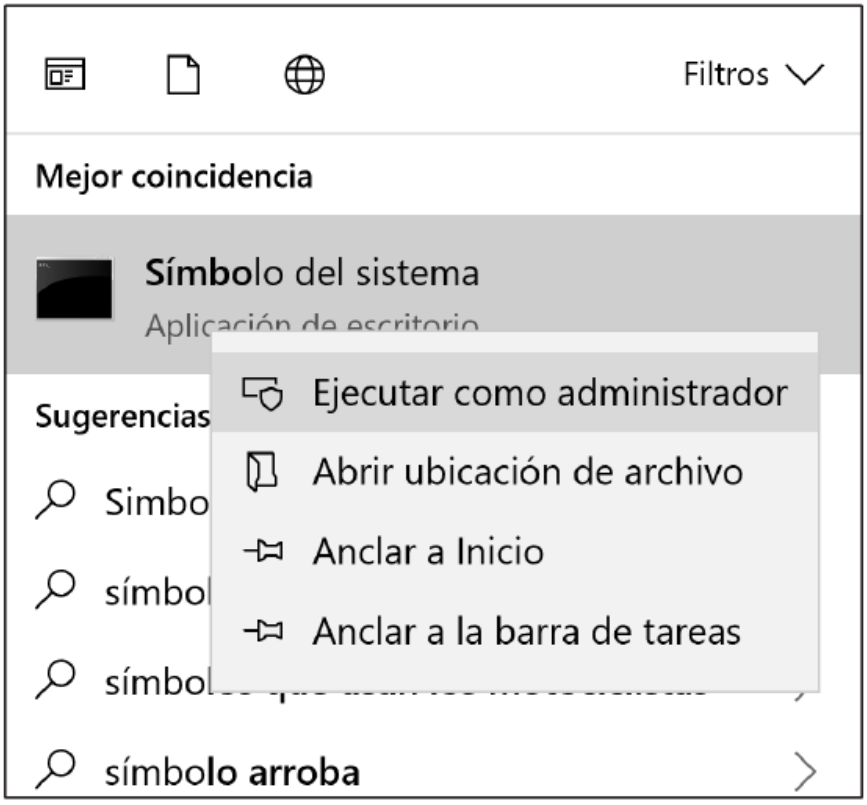


Figura A-7. Ejecutando el Símbolo del Sistema como Administrador.

Desde esta forma evitaremos posibles problemas de permisos durante la instalación.

Linux

Al igual que en caso de Windows, antes de ejecutar Spark en nuestro sistema debemos asegurarnos de que disponemos del entorno de ejecución de Java (JRE) instalado. Puede ser OpenJDK o la versión de Oracle, y se puede instalar de manera manual o a través de un gestor e paquetes (`apt`, `yum`), pero es importante que la variable de entorno `JAVA_HOME` esté convenientemente configurada para apuntar al directorio donde está instalado.

Una vez disponemos del JRE, para instalar Apache Spark en Linux deberemos acudir a la página de descargas

```
https://spark.apache.org/downloads.html
```

y seleccionar la versión deseada: para este libro será la 2.3.0 y el paquete "*Pre-built for Apache Hadoop 2.7 and later*". Una vez elegidas estas opciones, haremos clic en

el enlace `spark-2.3.0-bin-hadoop2.7.tgz` que aparece en el punto 3 del listado (ver figura A-4) para descargar el fichero `.tgz` en nuestro directorio local.

A continuación, descomprimimos el fichero, lo que creará una nueva carpeta en nuestro directorio local. Al igual que antes, renombraremos esa nueva carpeta a `spark` para facilitar futuras actualizaciones:

```
$ tar xvzf spark-2.3.0-bin-hadoop2.7.tgz
$ mv spark-2.3.0-bin-hadoop2.7 spark
```

Con esto ya tendríamos disponible Spark en nuestro sistema. Lo último que nos quedaría por hacer es incluir el directorio de binarios de Spark en el `PATH` para que sea más sencillo lanzarlo y configurar Spark para que utilice Jupyter al invocar al intérprete de Spark para Python (llamado PySpark):

```
# Ampliamos el PATH
export PATH=/home/<user>/spark/bin:$PATH

# Establecemos que pyspark se ejecutará desde Jupyter
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS=notebook
```

Abrimos un nuevo terminal para que se actualicen las variables de entorno y lanzamos PySpark mediante el comando:

```
$ pyspark
```

Si todo ha sido configurado correctamente, veremos algunos mensajes en la consola y automáticamente se abrirá el navegador predeterminado del sistema mostrando la pantalla principal de Jupyter, como en la figura A-2.

Mac OS

La instalación de Spark para Mac OS sigue los mismos pasos que los mostrados en la sección anterior para Linux. En el caso de Mac OS usamos Spark con el JRE de Eclipse.

ÍNDICE ANALÍTICO

$

$addToSet 102
$all 98
$and 95
$avg 102
$currentDate 111
$elemMatch 98
$eq 95
$exists 98
$first 102
$group 101
$gt 95
$gte 95
$in 98
$inc (update) 111
$last 102
$lookup 106
$lt 95
$lte 95
$match 103
$max ($group) 102
$max (update) 111
$min ($group) 102
$min (update) 111
$mul 111
$ne 95
$nin 98
$nor 95
$not 95
$or 95
$project 104
$push 102
$rename 111
$set 110
$setOnInsert 113
$size 98
$sortByCount 102
$sum 101
$unset 111
$unwind 104

&

  35

*

* 54

/

/ 52
// 55

[

[…] 56

_

_id 84
_setroot 21

<

… 36
… 35

 35
… 35
<form>…</form> 38
<h1>…</h1> 35
… 35
 36
<input> 38
… 36
… 35
<pre>…</pre> 35
… 35
<table>…</table> 37
<td>…</td> 37
<th>…</th> 37
<tr> …</tr> 37
**
…
** 36

A

acción (RDD) 159, 162
accuracy ..... *Véase* exactitud
add_sheet 10
agg 209
aggregate 100, 166
aggregateByKey 175
agregaciones 100
alias 207
align 236
análisis de grupos 129
Apache Spark 153
API 61, 72
API-REST ..... *Véase* REST
aprendizaje automático 126
aprendizaje no supervisado 129
aprendizaje supervisado 128
arange 227
array 15
Arrays 97
astype 125
atributo 127
atributo categórico 127
atributo clase 128
atributo continuo 128
Atributo nominal 127
Atributo ordinal 127
attrib 21
autopct 231
Axes 224
axis 226
Axis 224

B

bar 236
Bases de Datos NoSQL 79
bases de datos orientadas a documento 77
Bases de Datos Relacionales 79
BeautifulSoup 34
best 231
bins 241, 242
body 34
Book 9
bottom 225, 236, 242
boxes 234
boxplot 233

C

calinski_harabaz_score 145
caps 234
cartesian 176
cell 9
Cell 10
cellname 10
center 236

Ch

chromedriver 45

C

clasificación 128
clasificación binaria 129
clasificación multiclase 129
class 39
claves 84
cliente ..... *Véase* cliente-servidor

cliente-servidor 81
clustering ..... *Véase* análisis de grupos
ClusteringEvaluator 220
clustersCenters 220
coeficiente de silueta 132
`col` 10
colecciones 84
collect 163
`colname` 10
color 228, 236, 242
columns 120, 195
complex128 116
complex64 116
conjunto de datos distribuidos resilientes ..... *Véase* RDD
conjunto de entrenamiento 129
conjunto de test 129
conjunto de validación 131
contributors_enabled 67
coordinates 67
`count` 89, 95, 164
count (DataFrame Spark) 193
count (GroupedData) 209
created_at 66
createDataFrame 183
createOrReplaceTempView 210
`createView` ..... *Véase* vistas
cross validation .. *Véase* validación cruzada
CSV 2, 87
csv (DataFrameReader) 186
csv (DataFrameWriter) 188
`ctype` 10
cumulative 242

D

dashdot 227
dashed 227
Dask 151
Dask-ML 151
DataFrame 181
DataFrame (pandas) 117
DataFrameReader 186
dataframes 14
DataFrameWriter 188
DataSet 182
`default_profile` 67
DELETE 72
delimiter 7
derived 66
describe 122, 194
description 66
`DictReader` 5
DictWriter 6
distinct 175
dotted 227
driver 156
drop 86, 113, 195
drop (DataFrame pandas) 124
dropDuplicates 196
dropna 124, 196
DTD 20
dtypes 122
`dump` 16, 149
`dumps` 16

E

edge 236
`Element` 21
`ElementTree` 21
`encoding` 4
enlace ..... *Véase* link
`ensure_ascii` 17
entities 67, 68
equal 226
error absoluto medio 132
error *cuadrático medio* 132
escalado de atributos 135
esquema (Spark Dataframe) 184
etiqueta 19
etree 20, 22
exactitud 131
Excel 8
ExcelWriter 126
executor 156
explode 231
extended_entities 68

F

favorite_count 67
favorited 68

favourites_count 66
fieldnames 5
Figure 224
filter (DataFrame) 197
filter (RDD) 171
filter (Stream) 71
filter_level 68
find 21
find (ElementTree) 21
find (MongoDB) 89, 93
find_all 41
find_element_by_class_name 51
find_element_by_css_selector 51
find_element_by_id 51, 56
find_element_by_link_text 51
find_element_by_name 51
find_element_by_partial_link_text 51
find_element_by_tag_name 51
find_element_by_xpath 51, 56
find_one (MongoDB) 99
find_one (pymongo) 110
findall 21
findOne 99
fit 137
fit_transform 137
flatMap 171
flatMapValues 173
fliers 234
float16 116
float32 116
float64 116
follow 71
followers_count 66
format 191
Formula 11
friends_count 66
fromstring 21
fullOuterjoin 176
función kernel 140

G

gca 225
gcf 225
geckodriver 45
geo_enabled 67
geocode 69
GeoJSON 67
get 29, 72
GET 72
get_loc 139
get_text 41
getroot 21
Google Maps 72
GraphX 156
GridSearchCV 150
groupBy 208
groupByKey 173
GroupedData 208

H

hashtags 68
headless 59
height 236
hiperparámetro 130
hist 241
hspace 225
html.parser 35

I

id 39, 65, 66
id_str 65, 66
iloc 121
imputer 134
in_reply_to_screen_name 66
in_reply_to_status_id 65
in_reply_to_status_id_str 65
in_reply_to_user_id 65
in_reply_to_user_id_str 65
indent 17
índice Calinski-Harabasz 132
índices 92
insert 85
insert_many 110
insertOne 85
instancia 127
int16 116
int32 116
int64 116
int8 116

intersect....200
intersection....175
is_quote_status....67
ISODate....85
iter....21

J

joblib....148
join....106, 176, 200
json....72
JSON....15
json (DataFrameReader)....187
json (DataFrameWriter)....189

K

KMeans....145, 219
KNeighborsClassifier....141
KNeighborsRegressor....144

L

labels....231, 234
lang....67, 68, 69
last....57
left....225
leftOuterJoin....176
legend....231
limit....91
LinearRegression....143
LinearSVC....216
linestlye....228
link....47
listed_count....66
load....16, 149
loads....16
loc....121
locale....69
location....66
locations....66, 71
low....68

M

MAE....*Véase* error absoluto medio
map....169
MapReduce....153
mapValues....172
máquinas de vectores de soporte....140
marker....228
matchedCount....111
matching_rules....68
matplotlib....223, 224
matplotlib.pyplot....31
mean absolute error....*Véase mean squared errorVéase* error cuadrático medio
mean_absolute_error....143
mean_squared_error....143
means....234
media....68
Media....68
medians....234
medium....68
Mesos....155
MinMaxScaler....139
MinMaxScaler (SparkML)....216
missing values....*Véase* valores vacíos
MLlib....156
mode....191
mongo....82
MongoClient....86
mongod....82
MongoDB....77
mongoimport....87
MSE....*Véase* error cuadrático medio
muestreo....130
muestreo estratificado....130
MulticlassClassificationEvaluator....217

N

n 242
name....9, 66
ncols....9
ndarray....116
none....68
NoSuchElementException....51
notch....233
nrows....9
nsheets....9
null....103
NumPy....115

O

OAuthHandler 65
objetos 15
on_error 70
on_status 70
one hot encoding 134
OneHotEncoder 139
OneHotEncoderEstimator 215
open_workbook 9
option 191
orientation 236, 242
out 105

P

page 69
pandas 15
Pandas 117
parallelize 160
parse 21
particionador 158
patches 242
pickle 148
pie 230
Pipeline 147, 216
pipeline de agregación 101
place 67
plot 226, 227, 228
Poll 68
polls 68
position 57
positions 234
possibly_sensitive 68
POST 72
predict 137
preprocesado (aprendizaje automático) 133
pretty 90
printSchema 183
profile 67
protected 66
proyección 93
puerto 82
PUT 72
pymongo 81, 86, 99, 110
pyplot 223, 224

Q

q 69
quit 84
quote_count 67
quoted_status 67
quoted_status_id 67
quoted_status_id_str 67

R

random 88
randomSplit 214
range 242
RDD 157
RDD de parejas 158
read_csv 119
read_excel 119
reader 4
reduce 164
reduceByKey 174
register 204
regresión 129
regresión lineal 142
RegressionEvaluator 218
remove 113
replace_one 110
reply_count 67
requests 28, 72
Resilient Distributed Dataset ..... *Véase* RDD
Response 74
REST 72
retweet_count 67
retweeted 68
retweeted_status 67
right 225
rightOuterJoin 176
RMSE ..... *Véase* root mean squared error
root mean squared error 132
row 10
Row 22
rpp 69
Rule 68

S

sample 105
save 10

save (conector MongoDB) 191
save (Spark PipelineModel) 221
saveAsTextFile 168
savefig 228
sca 226
scaled 226
scf 225
scikit-learn 136
score 141
screen_name 66
search 69
`Search` 74
`select (BeautifulSoup)` 42
select (SparkSQL) 195, 206
selectExpr 203
`selenium` 44
`send_keys` 49
Series 117
servidor ..... *Véase* cliente-servidor
set_access_token 65
shadow 231
shape 120
`Sheet` 9
`sheet_by_index` 9
`sheet_by_name` 9
sheet_names() 9
`sheets` 9
`show collections` 108
`show databases` 83
show_user 70
silhouette_score 145
since_id 69
`size` 105
skip 90
`sort` 91
`sort_keys` 17
source 65
Spark ..... *Véase* Apache Spark
Spark SQL 155
Spark Streaming 155
spark.read 186
SparkContext 156
SparkML 156
SparkSession 182
sql 211
SQL 79
stage (Spark) 169
startangle 231
status_code 30
statuses_count 66
Stream 71
streaming 70
StreamListener 70
StringIndexer 214
`style` 39
`SubElement` 21
`submit` 38
subplot 225
subplots 224
subplots_adjust 225
subtract 176, 200
sum 209
support vector machines ..... *Véase* *SVM* ..... *Véase* máquinas de vectores de soporte
suptitle 226
SVC 140
sym 234
Symbol 68
symbols 68

T

tag 21
take 163
tasa de aciertos ..... *Véase* exactitud
text 21, 65
textFile 161
tick_label 236
tight 226
tight_layout 225
time_zone 66
Titanic, conjunto de datos 118
`title` 39, 226
to_csv 125
to_excel 126
token 63, 65
top 225
toPandas 190
track 71
train_test_split 138

transform 137
transformación (RDD) 159, 169
truncated 65
TSV 7
tubería scikit-learn 146
Tweepy 62
tweet 65

U

udf 207
UDF 204
uint16 116
uint32 116
uint64 116
uint8 116
union 175, 199
`update` 109
Update parcial 110
Update total 109
update_many 111
`update_one` 111
`updateMany` 111
`updateOne` 110
Upsert 112
URI 28, 72
url 66
URL 29, 68
urls 68
`use` 83
user 66
user_mentions 68
usermedians 234
UserMention 68
utc_offset 66

V

validación cruzada 130
valores vacíos 119
VectorAssembler 215
verified 66
vert 234
vistas 108

W

web scraping 27
`WebElement` 53
Wedge 231
`where` 94
whis 234
whiskers 234
whitheld_scope 67
wholeTextFiles 162
width 236
widths 234
withheld_in_countries 67
`Workbook` 10
`WorkSheet` 10
`write` 11, 21
`writeheader` 6
`writer` 6
`writerow` 5, 7
`writerows` 5
wspace 225

X

x 231, 233, 236, 241
xerror 236
xlabel 226
`xlrd` 8, 11
XLS ..... *Véase* EXCEL
xlwt 8, 10, 12
XML 19
XML Schema 20
XPath 21, 52
xticks 226

Y

YARN 155
yerror 236
ylabel 226
yticks 226