ÍNDICE

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 son:
  - 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. Deberemos 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