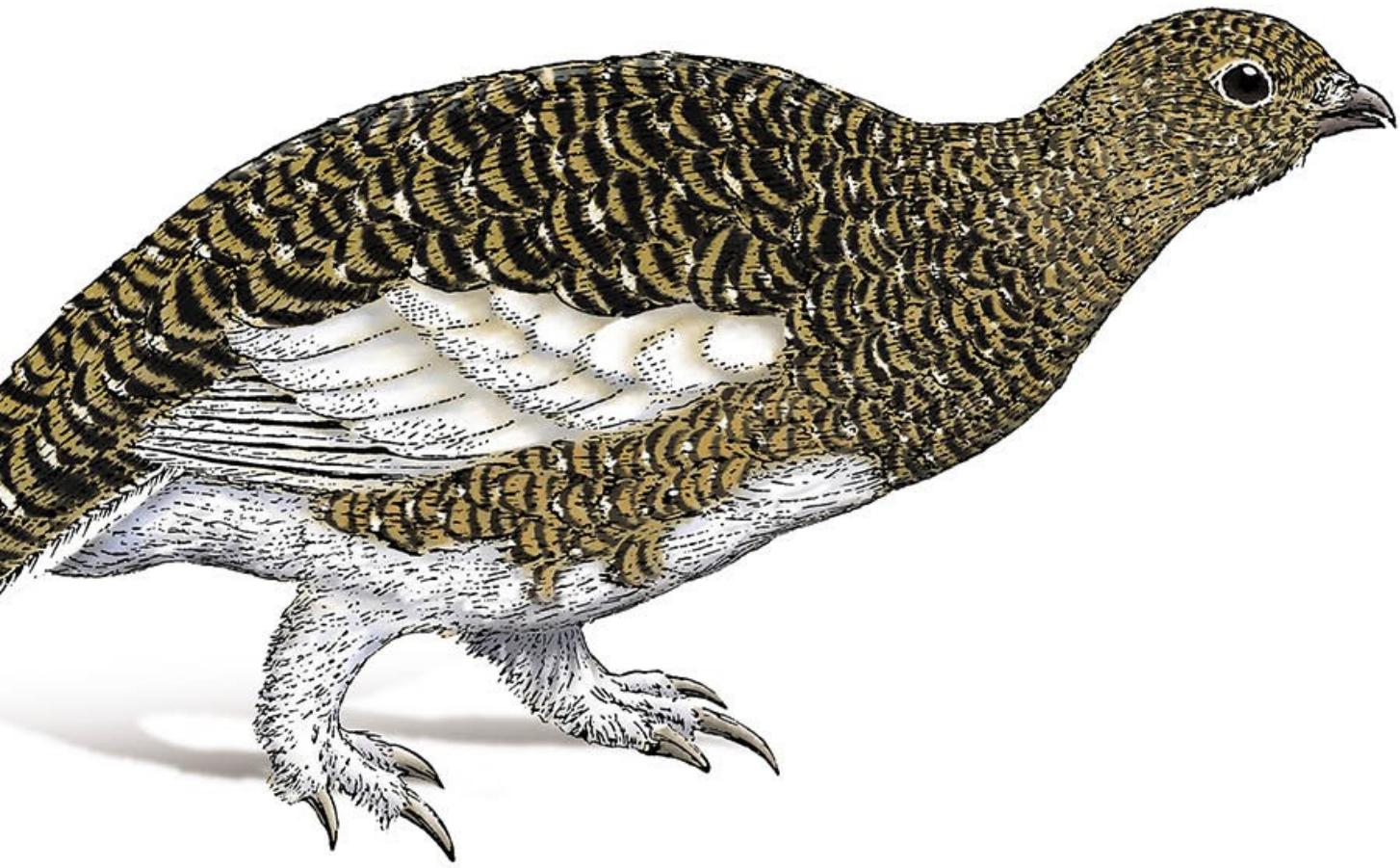
2.ª Edición

O'REILLY® ANAYA MULTIMEDIA

Ciencia de datos desde cero

Principios básicos con Python



Joel Grus

Ciencia de datos desde cero

Principios básicos con Python

2.ª Edición

Joel Grus

ANAYA MULTIMEDIA

Agradecimientos

En primer lugar, quiero agradecer a Mike Loukides por aceptar mi propuesta para este libro (y por insistir en que lo reduzca a un tamaño razonable). Habría sido muy fácil para él decir: "¿Quién es esta persona que no para de enviarme correos electrónicos con capítulos de muestra, y qué hago para deshacerme de él?". Pero me siento muy agradecido de que no lo dijera. También quisiera agradecer a mis editoras, Michele Cronin y Marie Beaugureau, por guiarme a lo largo del proceso de la publicación de libros y conseguir el libro en un estado mucho mejor de lo que jamás yo hubiera podido lograr por mí mismo.

No podría haber escrito este libro si nunca hubiera aprendido ciencia de datos o *data science*, y probablemente no habría aprendido ciencia de datos si no hubiera sido por la influencia de Dave Hsu, Igor Tatarinov, John Rauser y el resto de la banda Farecast (hace ya tanto tiempo que, en ese momento, la ciencia de datos ¡ni siquiera se conocía con ese nombre!). Hay que reconocerles también el gran mérito que tienen los buenos chicos de Coursera y DataTau.

Doy también las gracias a mis lectores y revisores beta. Jay Fundling encontró una gran cantidad de errores y resaltó muchas explicaciones no claras, y el libro es mucho mejor (y mucho más correcto) gracias a él. Debashis Shosh es un héroe por comprobar la sensatez de todas mis estadísticas. Andrew Musselman sugirió bajar el tono "la gente que prefiere R a Python son unos inmorales" del libro, lo que creo que finalmente acabó siendo un consejo bastante bueno. Trey Causey, Ryan Matthew Balfanz, Loris Mularoni, Núria Pujol, Rob Jefferson, Mary Pat Campbell, Zach Geary, Denise Mauldin, Jimmy O'Donnell y Wendy Grus proporcionaron también unos comentarios de gran valor. Gracias a todos los que leyeron la primera edición y ayudaron a que este libro fuera mejor. Los errores que puedan quedar son, por supuesto, responsabilidad mía.

Le debo mucho a la comunidad de Twitter #datascience por exponerme a un montón de conceptos nuevos, presentarme a mucha gente estupenda y

hacerme sentir tan manta, que se me ocurrió escribir un libro para compensarlo. Agradezco especialmente a Trey Causey (de nuevo) por recordarme (sin darse cuenta) incluir un capítulo sobre álgebra lineal y a Sean J. Taylor por señalar (sin darse cuenta) un par de enormes lapsus en el capítulo “Trabajando con datos”.

Por encima de todo, le debo una tonelada de agradecimientos a Ganga y Madeline. La única cosa más difícil que escribir un libro es vivir con alguien que esté escribiendo un libro, y no podría haberlo logrado sin su apoyo.

Sobre el autor

Joel Grus es ingeniero investigador en el Allen Institute for AI. Anteriormente trabajó como ingeniero de software en Google y como científico de datos en varias *startups*. Vive en Seattle, donde habitualmente asiste a *podcasts* sobre ciencia de datos. Tiene un blog que actualiza ocasionalmente en `joelgrus.com`, pero se pasa el día tuiteando en `@joelgrus`.

Índice

Prefacio a la segunda edición

Me siento excepcionalmente orgulloso de la primera edición de este libro. Ha resultado ser en buena parte el libro que yo quería que fuera. Pero varios años de desarrollos en ciencia de datos, de progreso en el ecosistema Python y de crecimiento personal como desarrollador y educador han cambiado lo que creo que debe ser un primer libro sobre ciencia de datos.

En la vida no hay vuelta atrás, pero en la escritura de libros sí hay segundas ediciones.

De acuerdo con esto, he reescrito todo el código y los ejemplos utilizando Python 3.6 (y muchas de sus funciones más recientes, como las anotaciones de tipos). En el libro hago continuamente énfasis en escribir código limpio. He reemplazado algunos de los ejemplos de la primera edición por otros más realistas, utilizando conjuntos de datos "reales". He añadido nuevo material en temas como *deep learning*, estadísticas y procesamiento del lenguaje natural, que se corresponden con cosas con las que es más probable que los científicos de datos de hoy en día trabajen (también he eliminado otras informaciones que parecían ser menos relevantes). Y he repasado el libro de una forma muy concienzuda, arreglando errores, reescribiendo explicaciones que eran menos claras de lo que podrían ser y actualizando algunos de los chistes.

La primera edición fue un gran libro, y esta edición es aún mejor. ¡Disfrútela!

Joel Grus
Seattle, WA
2019

Convenciones empleadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

- *Cursiva*: Es un tipo que se usa para diferenciar términos anglosajones o de uso poco común. También se usa para destacar algún concepto.
- **Negrita**: Le ayudará a localizar rápidamente elementos como las combinaciones de teclas.
- Fuente especial: Nombres de botones y opciones de programas. Por ejemplo, Aceptar para hacer referencia a un botón con ese título.
- `Monoespacial`: Utilizado para el código y dentro de los párrafos para hacer referencia a elementos como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

También encontrará a lo largo del libro recuadros con elementos destacados sobre el texto normal, comunicándole de manera breve y rápida algún concepto relacionado con lo que está leyendo, un truco o advirtiéndole de algo.

Aunque el termino *"data science"* es de uso generalizado y reconocido en todo el mundo, hemos decidido traducir este termino por "ciencia de datos" que es como se conoce a este área de conocimiento en castellano. Hemos preferido utilizar el termino en castellano por respeto a la riqueza de nuestra lengua y a los usuarios de los países de habla hispana.

Uso del código de ejemplo

Se puede descargar material adicional (ejemplos de código, ejercicios, etc.) de la página web de Anaya Multimedia (`http://www.anayamultimedia.es`). Vaya al botón Selecciona Complemento de la ficha del libro, donde podrá descargar el contenido para poder utilizarlo directamente. También puede descargar el material de la página web original del libro: `https://github.com/joelgrus/data-science-from-scratch`.

Este libro ha sido creado para ayudarle en su trabajo. En general, puede

utilizar el código de ejemplo incluido en sus programas y en su documentación. No es necesario contactar con la editorial para solicitar permiso a menos que esté reproduciendo una gran cantidad del código. Por ejemplo, escribir un programa que utilice varios fragmentos de código tomados de este libro no requiere permiso. Sin embargo, vender o distribuir un CD-ROM de ejemplos de los libros de O'Reilly sí lo requiere. Responder una pregunta citando este libro y empleando textualmente código de ejemplo incluido en él no requiere permiso. Pero incorporar una importante cantidad de código de ejemplo de este libro en la documentación de su producto sí lo requeriría.

Sobre la imagen de cubierta

El animal de la portada de este libro es una perdiz nival o lagópodo alpino (*Lagopus muta*). Este robusto miembro de la familia de los faisánidos, del tamaño de un pollo, vive en la tundra del hemisferio norte, en las regiones árticas y subárticas de Eurasia y Norteamérica. Se alimenta de lo que encuentra en el suelo, recorriendo las praderas con sus patas bien emplumadas, comiendo brotes de abedul y sauce, así como semillas, flores, hojas y bayas. Los polluelos de perdiz nival también comen insectos.

Los lagópodos alpinos son muy conocidos por los sorprendentes cambios anuales que sufre su enigmático camuflaje, habiendo evolucionado para mudar sus plumas blancas y pardas varias veces en el transcurso de un año y así adaptarse mejor a los cambiantes colores estacionales de su entorno. En invierno tienen plumas blancas; en primavera y otoño, cuando el manto nevado se mezcla con la dehesa, su plumaje mezcla los colores blanco y pardo y, en verano, sus plumas, pardas por completo, coinciden con la variada coloración de la tundra. Con este camuflaje, las hembras pueden incubar sus huevos, que dejan en nidos sobre el suelo, siendo casi invisibles.

Las perdices nivales macho adultas tienen también una especie de cresta roja sobre sus ojos. Durante la temporada de cría la utilizan para el cortejo, así como en los combates contra otros machos (existen estudios que demuestran una correlación entre el tamaño de la cresta y el nivel de

testosterona de los machos).

La población de perdiz de las nieves está actualmente en declive, aunque en su hábitat natural siguen siendo comunes (pero difíciles de observar). Tienen muchos depredadores, entre otros el zorro ártico, el gerifalte, la gaviota común y la gaviota salteadora o escua. Además, con el tiempo, el cambio climático puede afectar negativamente a sus cambios de color estacionales.

Muchos de los animales de las portadas de O'Reilly están en peligro de extinción; todos ellos son importantes para el mundo.

La imagen de la portada procede de la obra *Cassell's Book of Birds* (1875), de Thomas Rymer Jones.

Prefacio a la primera edición

Ciencia de datos o data science

El trabajo de científico de datos ha sido denominado “el empleo más sexy del siglo XXI”,[1] presuntamente por alguien que no ha visitado nunca un parque de bomberos. Sin embargo, la ciencia de datos es un campo en pleno auge y crecimiento, y no hace falta ser muy perspicaz para encontrar analistas prediciendo sin descanso que, en los próximos 10 años, necesitaremos miles de millones de científicos de datos más de los que tenemos ahora.

Pero ¿qué es la ciencia de datos? Después de todo, no podemos crear científicos de datos si no sabemos cuál es su trabajo. Según un diagrama de Venn,[2] que es en cierto modo famoso en este sector, la ciencia de datos reside en la intersección entre:

- Habilidades informáticas a nivel de *hacker*.
- Conocimiento de matemáticas y estadística.
- Experiencia relevante.

Aunque mi intención inicial era escribir un libro que hablara sobre estos tres puntos, rápidamente me di cuenta de que un tratamiento en profundidad de la expresión “experiencia relevante” requeriría cientos de miles de páginas. En ese momento decidí centrarme en los dos primeros. Mi objetivo es ayudar a los lectores a desarrollar las habilidades informáticas a nivel de *hacker* que necesitarán para empezar a trabajar en la ciencia de datos. Pero también es permitirles sentirse cómodos con las matemáticas y la estadística, que son el núcleo de la ciencia de datos.

Quizá esta sea una aspiración demasiado elevada para un libro. La mejor forma de aprender las habilidades informáticas de un *hacker* es hackeando cosas. Leyendo este libro, los lectores podrán llegar a comprender bastante

bien la forma en la que yo hackeo cosas, que no tiene por qué ser necesariamente la suya. También conocerán bastante bien algunas de las herramientas que utilizo, que no han de ser obligadamente las mejores para ellos. Y entenderán bien el modo en que yo abordo los problemas de datos, que tampoco tiene por qué ser el mejor modo para ellos. La intención (y la esperanza) es que mis ejemplos les inspiren a probar las cosas a su manera. Todo el código y los datos del libro están disponibles en GitHub[3] para que puedan ponerse manos a la obra.

De forma similar, la mejor manera de aprender matemáticas es haciendo matemáticas. Este no es rotundamente un libro de mates, y en su mayor parte no estaremos "haciendo matemáticas". Sin embargo, no se puede hacer ciencia de datos de verdad sin ciertos conocimientos de probabilidad, estadística y álgebra lineal. Esto significa que, donde corresponda, profundizaremos en ecuaciones matemáticas, intuición matemática, axiomas matemáticos y versiones caricaturizadas de grandes ideas matemáticas. Espero que los lectores no teman sumergirse conmigo.

A lo largo de todo el libro también espero dar a entender que jugar con datos es divertido porque, bueno, ¡jugar con datos realmente lo es! (especialmente si lo comparamos con algunas alternativas, como hacer la declaración de la renta o trabajar en una mina).

Partir de cero

Hay muchísimas librerías de ciencia de datos, *frameworks*, módulos y kits de herramientas que implementan de forma eficaz los algoritmos y las técnicas de ciencia de datos más conocidas (así como las menos habituales). Si alguno de mis lectores llega a ser científico de datos, acabará estando íntimamente familiarizado con NumPy, scikit-learn, pandas y todas las demás librerías existentes. Son fabulosas para hacer ciencia de datos, pero también suponen una buena forma de empezar a hacer ciencia de datos sin realmente comprender lo que es.

En este libro nos acercaremos a la ciencia de datos desde el principio de los principios. Esto significa que crearemos herramientas e implementaremos

algoritmos a mano para poder comprenderlos mejor. Pensé mucho en crear implementaciones y ejemplos que fueran claros y legibles y estuvieran bien comentados. En la mayoría de los casos, las herramientas que construiremos serán esclarecedoras, pero poco prácticas. Funcionarán bien en pequeños conjuntos de datos, pero no lo harán en otros "a gran escala". Durante todo el libro iré señalando las librerías que se podrían utilizar para aplicar estas técnicas sobre conjuntos de datos más grandes. Pero aquí no las utilizaremos.

Existe una sana discusión en torno al mejor lenguaje que se puede utilizar para aprender ciencia de datos. Mucha gente cree que es el lenguaje de programación estadístico R (de esas personas decimos que están equivocadas). Unas pocas personas sugieren Java o Scala. Sin embargo, en mi opinión, Python es la elección obvia.

Python tiene varias características que le hacen ser ideal para aprender (y hacer) ciencia de datos:

- Es gratuito.
- Es relativamente sencillo para crear código (y en particular, de comprender).
- Tiene muchas librerías asociadas a la ciencia de datos que son de gran utilidad.

Dudo si decir que Python es mi lenguaje de programación favorito. Hay otros lenguajes que encuentro más agradables, mejor diseñados o simplemente más divertidos de utilizar. Pero, aun así, cada vez que inicio un nuevo proyecto de ciencia de datos, termino utilizando Python. Cada vez que necesito crear rápidamente un prototipo de algo que simplemente funcione, termino utilizando Python. Y cada vez que quiero demostrar conceptos de ciencia de datos de una forma clara y sencilla de comprender, acabo por utilizar Python. De ahí que este libro utilice Python.

El objetivo de este libro no es enseñar Python (aunque es casi seguro que leyéndolo se aprende un poco). Llevaré a los lectores a lo largo de un curso acelerado de un capítulo de duración, que resalta las características más importantes para nuestros propósitos, pero, si no se sabe nada sobre programar en Python (o sobre programar en general), entonces quizá

convenga complementar este libro con algún tutorial de tipo “Python para principiantes”.

El resto de nuestra introducción a la ciencia de datos seguirá este mismo enfoque, es decir, entrar en detalle donde ello parezca ser crucial o esclarecedor, pero en otras ocasiones dejarle al lector los detalles para que investigue por sí mismo (o lo busque en la Wikipedia).

Durante años he formado a un buen número de científicos de datos. Aunque no todos han evolucionado para convertirse en revolucionarias estrellas del rock ninja de los datos, les he dejado siendo mejores científicos de datos de lo que eran cuando les conocí. Y yo he llegado a creer que cualquiera que tenga una cierta cantidad de aptitud matemática y una determinada habilidad para programar tiene los fundamentos necesarios para hacer ciencia de datos. Todo lo que se necesita es una mente curiosa, voluntad para trabajar duro y este libro. De ahí este libro.

---

[1] https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century.

[2] http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram.

[3] https://github.com/joelgrus/data-science-from-scratch.

1 Introducción

"¡Datos, datos, datos!", gritó con impaciencia. "No puedo hacer ladrillos sin arcilla".
—Arthur Conan Doyle

El ascenso de los datos

Vivimos en un mundo que se está ahogando en datos. Los sitios web controlan cada clic de cada usuario. Los teléfonos inteligentes crean registros con la ubicación y velocidad de sus dueños cada segundo de cada día. Cada vez más autodidactas llevan puestos podómetros superpotentes que registran continuamente su ritmo cardíaco, sus hábitos de movimiento, su dieta y sus patrones de sueño. Los coches inteligentes recogen las costumbres en la conducción, las casas inteligentes recopilan hábitos de vida y los comerciantes inteligentes recolectan hábitos de compra. Internet representa un grafo de conocimiento gigante que contiene (entre otras cosas) una enorme enciclopedia con referencias cruzadas, bases de datos específicas de dominio sobre películas, música, resultados deportivos, máquinas de *pinball*, memes y cócteles, y tal cantidad de estadísticas gubernamentales (¡algunas de ellas casi ciertas!) de tantos gobiernos que no le caben a uno en la cabeza.

Enterradas en estos datos están las respuestas a incontables preguntas que nadie nunca pensó en responder. En este libro aprenderemos a encontrarlas.

¿Qué es la ciencia de datos o data science?

Hay un chiste que dice que un científico de datos es alguien que conoce más estadísticas que un científico informático y más ciencia informática que un estadista (yo no diría que el chiste es bueno). De hecho, algunos

científicos de datos son (a todos los efectos prácticos) estadistas, mientras que a otros apenas se les puede distinguir de un ingeniero de software. Algunos son expertos en *machine learning*, mientras que otros no llegaron a aprender ni tan siquiera por dónde se salía de la guardería. Algunos son doctores con impresionantes registros de publicaciones, mientras que otros nunca han leído un documento académico (aunque les dé vergüenza admitirlo). En resumen, da igual cómo se defina la ciencia de datos; siempre habrá profesionales para los que la definición es total y absolutamente errónea. Sin embargo, no dejaremos que eso nos impida seguir intentándolo. Diremos que un científico de datos es alguien que extrae conocimientos a partir de datos desordenados. El mundo de hoy en día está repleto de personas que intentan convertir datos en conocimiento.

Por ejemplo, en el sitio web de citas OkCupid se les pide a sus miembros que respondan a cientos de preguntas para poder encontrar las parejas más adecuadas para ellos. Pero también analizan estos resultados para dar con preguntas aparentemente inocuas que se le puedan formular a alguien para averiguar la probabilidad de que esa persona vaya a dormir contigo en la primera cita.[1]

Facebook suele preguntar a sus usuarios por su ciudad natal y su ubicación actual, aparentemente para que les resulte más fácil a sus amigos encontrarles y conectar con ellos. Pero también analiza estas ubicaciones para identificar patrones de migración globales[2] y para averiguar dónde viven las comunidades de aficionados de distintos equipos de fútbol.[3]

La cadena americana de grandes almacenes Target controla las compras e interacciones de sus usuarios, tanto en línea como en sus tiendas físicas. Igualmente realiza modelos predictivos[4] con los datos para averiguar cuáles de sus clientas están embarazadas y así venderles con más facilidad productos relacionados con el bebé.

En 2012, la campaña de Obama empleó a muchísimos científicos que investigaron con datos y experimentaron con ellos para identificar a los votantes que necesitaban más atención, eligiendo los métodos perfectos de recaudación de fondos dirigidos a determinados donantes y poniendo todos los esfuerzos de obtención del voto allí donde era más probable que fueran útiles. En 2016, la campaña Trump probó una asombrosa variedad de

anuncios *online*[5] y analizó los datos para averiguar lo que funcionaba y lo que no. Lo último antes de resultar cansino: algunos científicos de datos utilizan también de vez en cuando sus habilidades para cosas buenas, por ejemplo, para que el gobierno sea más eficaz[6] o para ayudar a los sin techo.[7] Pero sin duda tampoco resultan perjudicados si lo que les gusta es buscar la mejor manera de conseguir que la gente haga clic en anuncios.

Hipótesis motivadora: DataSciencester

¡Felicidades! Le acaban de contratar para dirigir el departamento de Ciencia de Datos de DataSciencester, la red social para científicos de datos.

> **Nota:** Cuando escribí la primera edición de este libro, pensé que "una red social para científicos de datos" era una invención que podía resultar absurda, pero a la vez divertida. Desde entonces, se han creado redes sociales para científicos de datos de verdad, y sus creadores les han sacado a capitalistas de riesgo mucho más dinero del que yo obtuve con mi libro. Probablemente esto suponga una valiosa lección sobre inventos absurdos de ciencia de datos o la publicación de libros.

A pesar de estar destinada a científicos de datos, DataSciencester nunca ha invertido realmente en crear sus propios métodos de ciencia de datos (para ser justo, nunca ha invertido realmente en crear siquiera su producto). Esto será lo que hagamos aquí. A lo largo del libro, aprenderemos conceptos de ciencia de datos resolviendo problemas con los que uno se puede encontrar en el trabajo. Algunas veces veremos datos suministrados específicamente por los usuarios, otras veces examinaremos datos generados por sus interacciones con sitios web, y en otras ocasiones incluso trataremos datos de experimentos que nosotros mismos diseñaremos.

Como DataSciencester sufre terriblemente el síndrome NIH (*Not invented here*, no inventado aquí), crearemos nuestras propias herramientas desde cero. Al final, el lector terminará comprendiendo bastante bien los fundamentos de la ciencia de datos y podrá aplicar sus habilidades en una

compañía con una premisa menos inestable o en cualquier otro problema que le interese.

Bienvenidos a bordo y ¡buena suerte! (se pueden llevar vaqueros los viernes y el baño está al fondo a la derecha).

Localizar los conectores clave

Es el primer día de trabajo en DataSciencester, y el vicepresidente de Redes tiene muchas preguntas sobre los usuarios. Hasta ahora no tenía nadie a quien preguntar, así que está muy emocionado de tener alguien nuevo en el equipo.

En particular, le interesa identificar quiénes son los "conectores clave" de todos los científicos de datos. Para ello proporciona un volcado de la red completa de DataSciencester (en la vida real, la gente no suele pasar los datos que uno necesita; el capítulo 9 está dedicado a obtener datos).

¿Qué aspecto tiene este volcado de datos? Consiste en una lista de usuarios, cada uno representado por un `dict` que contiene su `id` (que es un número) y su `name` (que, en una de esas fabulosas conjunciones planetarias, concuerda con su `id`):

```
users = [
   { "id": 0, "name": "Hero" },
   { "id": 1, "name": "Dunn" },
   { "id": 2, "name": "Sue" },
   { "id": 3, "name": "Chi" },
   { "id": 4, "name": "Thor" },
   { "id": 5, "name": "Clive" },
   { "id": 6, "name": "Hicks" },
   { "id": 7, "name": "Devin" },
   { "id": 8, "name": "Kate" },
   { "id": 9, "name": "Klein" }
]
```

También ofrece los datos de "amistad" (*friendship*), representados como una lista de pares de identificadores:

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

Por ejemplo, la tupla (0, 1) indica que los científicos de datos con `id` 0 (Hero) e `id` 1 (Dunn) son amigos. La red aparece representada en la figura 1.1.

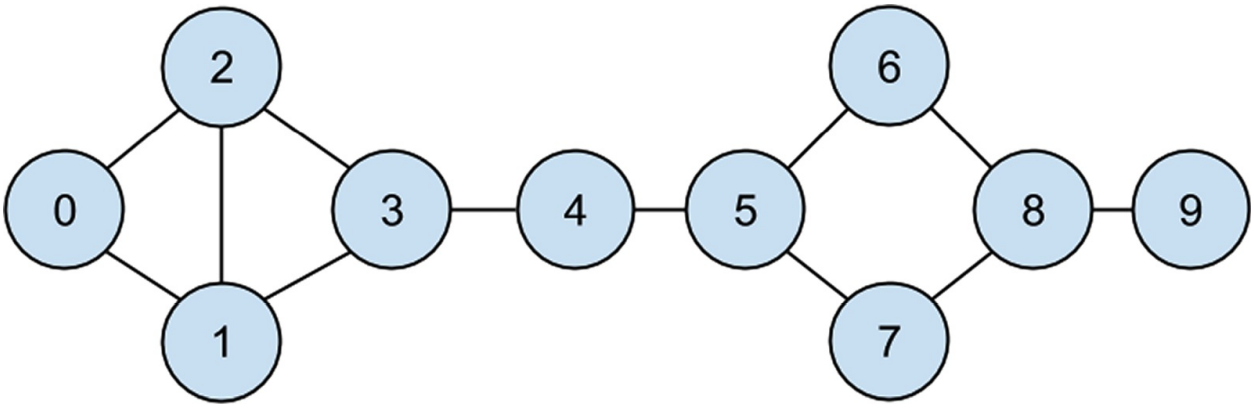


Figura 1.1. La red de DataSciencester.

Representar las amistades como una lista de pares no es la forma más sencilla de trabajar con ellas. Para encontrar todas las amistades por usuario, hay que pasar repetidamente por cada par buscando pares que contengan 1. Si hubiera muchos pares, el proceso tardaría mucho en realizarse.

En lugar de ello, vamos a crear un `dict` en el que las claves sean `id` de usuario y los valores sean listas de `id` de amigos (consultar cosas en un `dict` es muy rápido).

> **Nota:** No conviene obsesionarse demasiado con los detalles del código ahora mismo. En el capítulo 2 haremos un curso acelerado de Python. Por ahora, basta con hacerse una idea general de lo que estamos haciendo.

Aún tendremos que consultar cada par para crear el `dict`, pero solamente hay que hacerlo una vez y, después, las consultas no costarán nada:

```
# Inicializar el dict con una lista vacía para cada id de usuario:
friendships = {user["id"]: [] for user in users}
# Y pasar por todos los pares de amistad para llenarlo:
```

```
for i, j in friendship_pairs:
   friendships[i].append(j)     # Añadir j como un amigo del usuario i
   friendships[j].append(i)     # Añadir i como un amigo del usuario j
```

Ahora que ya tenemos las amistades en un `dict`, podemos formular fácilmente preguntas sobre nuestro grafo, como por ejemplo: "¿Cuál es el número medio de conexiones?".

Primero, hallamos el número total de conexiones sumando las longitudes de todas las listas `friends`:

```
def number_of_friends(user):
   """How many friends does _user_ have?"""
   user_id = user["id"]
   friend_ids = friendships[user_id]
   return len(friend_ids)
total_connections = sum(number_of_friends(user)
                  for user in users)                # 24
```

Y, después, simplemente dividimos por el número de usuarios:

```
num_users = len(users)                              # longitud de la lista de
                                                    usuarios
avg_connections = total_connections /               # 24 / 10 == 2,4
num_users
```

También es sencillo encontrar las personas más conectadas (las que tienen la mayor cantidad de amigos).

Como no hay muchos usuarios, simplemente podemos ordenarlos de "la mayor cantidad de amigos" a "la menor cantidad de amigos":

```
# Crea una lista (user_id, number_of_friends).
num_friends_by_id = [(user["id"], number_of_friends(user))
                for user in users]
num_friends_by_id.sort(                                    # Ordena la lista
     key=lambda id_and_friends: id_and_friends[1],         # por num_friends
     reverse=True)                                         # del mayor al menor
# Cada par es (user_id, num_friends):
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
# (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

Una manera de pensar en lo que hemos hecho es como en una forma de identificar a las personas que son de alguna manera centrales para la red. En realidad, lo que acabamos de calcular es la métrica de la centralidad de grado de la red (véase la figura 1.2).

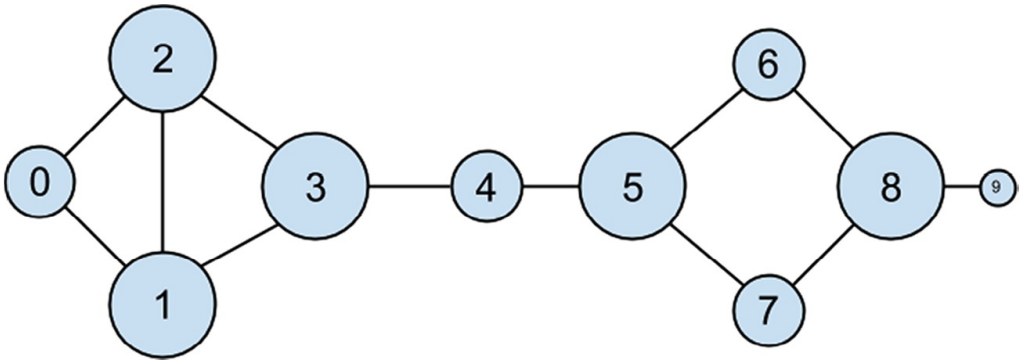


Figura 1.2. La red de DataSciencester dimensionada por grado.

Esto tiene la virtud de ser bastante fácil de calcular, pero no siempre da los resultados que se desean o esperan. Por ejemplo, en la red de DataSciencester Thor (`id` 4) solo tiene dos conexiones, mientras que Dunn (`id` 1) tiene tres. Ya cuando miramos a la red, parece intuitivo que Thor debería estar ubicado en una posición más central. En el capítulo 22 investigaremos las redes con más detalle y veremos nociones más complejas de centralidad, que pueden corresponderse más o menos con nuestra intuición.

Científicos de datos que podría conocer

Mientras aún está rellenando el papeleo de su nueva contratación, la vicepresidenta de Fraternización pasa por su despacho. Quiere fomentar más conexiones entre sus miembros y le pide que diseñe un sugeridor “Científicos de datos que podría conocer”.

Lo primero que se le ocurre es sugerir que los usuarios podrían conocer a los amigos de sus amigos. Así que escribe un poco de código para pasar varias veces por sus amigos y recoger los amigos de los amigos:

```
def foaf_ids_bad(user):
```

```
    """foaf is short for "friend of a friend" """
    return [foaf_id
         for friend_id in friendships[user["id"]]
         for foaf_id in friendships[friend_id]]
```

Cuando aplicamos esto sobre `users[0]` (Hero), produce lo siguiente:

```
[0, 2, 3, 0, 1, 3]
```

Incluye el usuario 0 dos veces, ya que Hero es de hecho amigo de sus dos amigos. Incluye los usuarios 1 y 2, aunque ambos ya son amigos de Hero. Y también incluye el usuario 3 dos veces, ya que se puede llegar hasta Chi a través de dos amigos distintos:

```
print(friendships[0]) # [1, 2]
print(friendships[1]) # [0, 2, 3]
print(friendships[2]) # [0, 1, 3]
```

Saber que las personas son amigos de amigos de diversas maneras parece ser información interesante, de modo que quizá en su lugar podríamos producir un contador de amigos mutuos. Y deberíamos probablemente excluir gente ya conocida por el usuario:

```
from collections import Counter                  # no cargado inicialmente
def friends_of_friends(user):
   user_id = user[“id”]
   return Counter(
      foaf_id
      for friend_id in friendships[user_id]      # Para cada uno de mis amigos,
      for foaf_id in friendships[friend_id]      # encuentra sus amigos
      if foaf_id != user_id                      # que no son yo
      and foaf_id not in                         # y no son mis amigos
      friendships[user_id]
   )
print(friends_of_friends(users[3]))              # Contador({0: 2, 5: 1})
```

Esto le dice correctamente a Chi (`id` 3) que tiene dos amigos mutuos con Hero (`id` 0), pero solo uno con Clive (`id` 5).

Uno, como científico de datos, sabe que también se puede disfrutar conociendo amigos con intereses comunes (este es un buen ejemplo del

aspecto “experiencia relevante” de la ciencia de datos). Tras preguntar por ahí, conseguimos estos datos, como una lista de pares (`user_id`, `interest`):

```
interests = [
   (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
   (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
   (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
   (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
   (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
   (3, "statistics"), (3, "regression"), (3, "probability"),
   (4, "machine learning"), (4, "regression"), (4, "decision trees"),
   (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
   (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
   (6, "probability"), (6, "mathematics"), (6, "theory"),
   (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
   (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
   (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
   (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

Por ejemplo, Hero (`id` 0) no tiene amigos comunes con Klein (`id` 9), pero ambos comparten intereses en Java y Big Data.

Es fácil crear una función que encuentre usuarios con un determinado interés:

```
def data_scientists_who_like(target_interest):
   """Find the ids of all users who like the target interest."""
   return [user_id
      for user_id, user_interest in interests
      if user_interest == target_interest]
```

Esto funciona, pero tiene que examinar la lista completa de aficiones en cada búsqueda. Si tenemos muchos usuarios e intereses (o si simplemente queremos hacer muchas búsquedas), es probablemente mejor que nos dediquemos a crear un índice de intereses a usuarios:

```
from collections import defaultdict
# Las claves son intereses, los valores son listas de user_ids con ese interés
user_ids_by_interest = defaultdict(list)
for user_id, interest in interests:
```

```
    user_ids_by_interest[interest].append(user_id)
```

Y otro de usuarios a intereses:

```
# Las claves son user_ids, los valores son listas de intereses para ese
user_id.
interests_by_user_id = defaultdict(list)
for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

Ahora es fácil averiguar quién tiene el mayor número de intereses en común con un determinado usuario:

- Pasamos varias veces por los intereses del usuario.
- Para cada interés, volvemos a pasar en repetidas ocasiones por los demás usuarios que tienen ese mismo interés.
- Contamos las veces que vemos cada uno de los usuarios.

En código:

```
def most_common_interests_with(user):
    return Counter(
        interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"]
)
```

Después, podríamos utilizar esto para crear una función “Científicos de datos que podría conocer” más completa basándonos en una combinación de amigos mutuos e intereses comunes. Exploraremos estos tipos de aplicación en el capítulo 23.

Salarios y experiencia

Justo cuando se iba a comer, el vicepresidente de Relaciones Públicas le

pregunta si le puede suministrar datos curiosos sobre lo que ganan los científicos de datos. Los datos de sueldos son, por supuesto, confidenciales, pero se las arregla para conseguir un conjunto de datos anónimo que contiene el salario (`salary`) de cada usuario (en dólares) y su antigüedad en el puesto (`tenure`) como científico de datos (en años):

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
           (48000, 0.7), (76000, 6),
           (69000, 6.5), (76000, 7.5),
           (60000, 2.5), (83000, 10),
           (48000, 1.9), (63000, 4.2)]
```

El primer paso natural es trazar los datos en un gráfico (cosa que veremos cómo hacer en el capítulo 3). La figura 1.3 muestra los resultados.

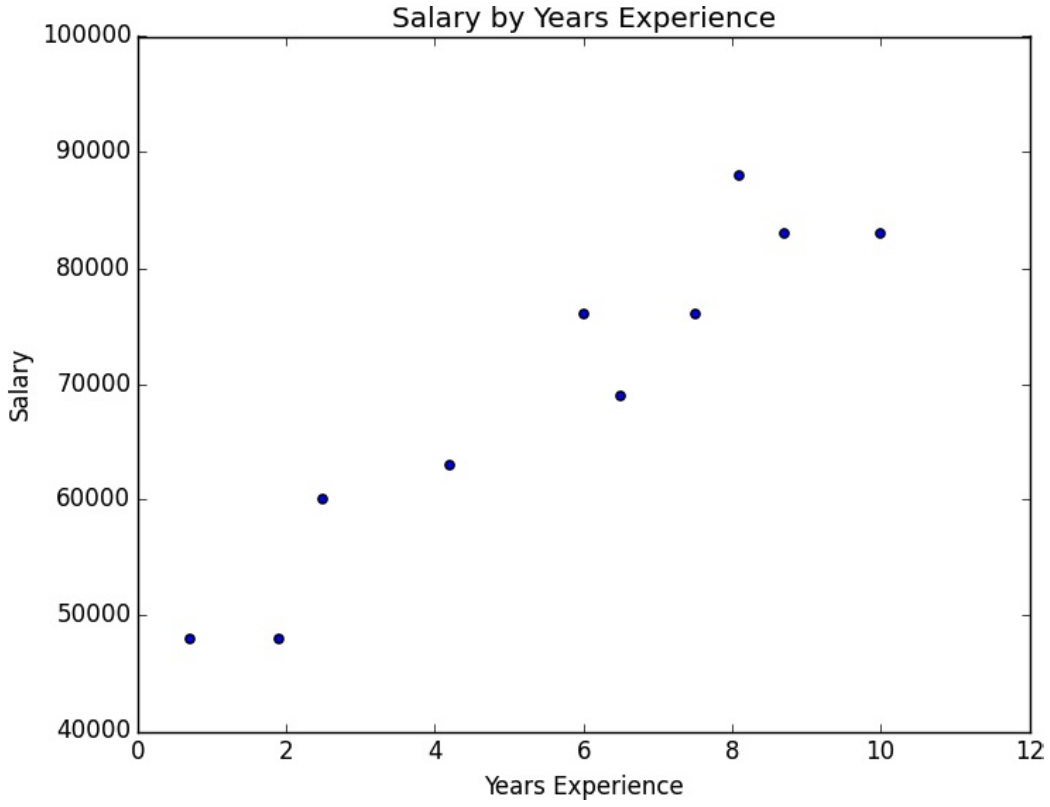


Figura 1.3. Salario por años de experiencia.

Parece claro que la gente con más experiencia tiende a ganar más. ¿Cómo se puede convertir esto en un dato curioso? Lo primero que se nos ocurre es

mirar el salario medio por antigüedad:

```
# Las claves son años, los valores son listas de los salarios por antigüedad.
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
   salary_by_tenure[tenure].append(salary)

# Las claves son años, cada valor es el salario medio para dicha antigüedad.
average_salary_by_tenure = {
   tenure: sum(salaries) / len(salaries)
   for tenure, salaries in salary_by_tenure.items()
}
```

Resulta que esto no es especialmente útil, ya que ninguno de los usuarios tiene la misma antigüedad en el puesto de trabajo, lo que significa que simplemente estamos informando de los salarios individuales de los usuarios:

```
{0.7: 48000.0,
   1.9: 48000.0,
   2.5: 60000.0,
   4.2: 63000.0,
   6: 76000.0,
   6.5: 69000.0,
   7.5: 76000.0,
   8.1: 88000.0,
   8.7: 83000.0,
   10: 83000.0}
```

Podría ser más útil poner los años de antigüedad en un *bucket*:

```
def tenure_bucket(tenure):
   if tenure < 2:
      return "less than two"
   elif tenure < 5:
      return "between two and five"
   else:
      return "more than five"
```

Entonces podemos agrupar los salarios correspondientes a cada *bucket*:

```
# Las claves son buckets de años de antigüedad, los valores son listas de
salarios para bucket
salary_by_tenure_bucket = defaultdict(list)
for salary, tenure in salaries_and_tenures:
   bucket = tenure_bucket(tenure)
   salary_by_tenure_bucket[bucket].append(salary)
```

Y, por último, calcular el salario medio para cada grupo:

```
# Las claves son buckets de años de antigüedad, los valores son el salario
medio para bucket
average_salary_by_bucket = {
   tenure_bucket: sum(salaries) / len(salaries)
   for tenure_bucket, salaries in salary_by_tenure_bucket.items()
}
```

Lo que es más interesante:

```
{‘between two and five’: 61500.0,
   ‘less than two’: 48000.0,
   ‘more than five’: 79166.66666666667}
```

Y ya tenemos nuestra proclama: “Los científicos de datos con más de cinco años de experiencia ganan un 65 % más que los científicos de datos con poca experiencia o ninguna”.

Pero hemos elegido los *buckets* de una forma bastante aleatoria. Lo que realmente haríamos es hacer alguna declaración sobre el efecto que tiene en el salario (en promedio) tener un año adicional de experiencia. Además de conseguir un dato curioso más eficiente, esto nos permite hacer predicciones sobre salarios que no conocemos. Exploraremos esta idea en el capítulo 14.

Cuentas de pago

Cuando vuelve a su puesto de trabajo, la vicepresidenta de Finanzas le está esperando. Quiere entender mejor qué usuarios pagan por las cuentas y cuáles no (ella conoce sus nombres, pero esa información no es especialmente procesable).

Se da cuenta de que parece haber una correspondencia entre los años de experiencia y las cuentas de pago:

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6.0 unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10.0 paid
```

Los usuarios con muy pocos y muchos años de experiencia tienden a pagar; los usuarios con cantidades de experiencia medias no lo hacen. Según esto, si quería crear un modelo (aunque sin duda no son datos suficientes en los que basarlo), podría intentar predecir “de pago” para usuarios con muy pocos y muchos años de experiencia y “no de pago” para usuarios con cantidades de experiencia medias:

```
def predict_paid_or_unpaid(years_experience):
   if years_experience < 3.0:
      return "paid"
   elif years_experience < 8.5:
      return "unpaid"
   else:
      return "paid"
```

Por supuesto, esto lo hemos calculado a ojo.

Con más datos (y más matemáticas), podríamos crear un modelo que predijera la probabilidad de que un usuario pagara, basándonos en sus años de experiencia. Investigaremos este tipo de problema en el capítulo 16.

Temas de interés

A medida que se va acercando el final de su primer día, la vicepresidenta

de Estrategia de Contenidos le pide datos sobre los temas que más interesan a los usuarios, de forma que pueda planificar adecuadamente el calendario de su blog. Ya disponemos de los datos sin procesar del proyecto del sugeridor de amigos:

```
interests = [
   (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
   (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
   (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
   (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
   (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
   (3, "statistics"), (3, "regression"), (3, "probability"),
   (4, "machine learning"), (4, "regression"), (4, "decision trees"),
   (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
   (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
   (6, "probability"), (6, "mathematics"), (6, "theory"),
   (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
   (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
   (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
   (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

Una manera sencilla (aunque no especialmente apasionante) de encontrar los intereses más populares es contando las palabras:

1. Ponemos en minúsculas todos los *hobbies* (ya que habrá usuarios que los pongan en mayúscula y otros en minúscula).
2. Los dividimos en palabras.
3. Contamos los resultados.

En código:

```
words_and_counts = Counter(word
                   for user, interest in interests
                   for word in interest.lower().split())
```

Así es posible hacer fácilmente un listado con las palabras que aparecen más de una vez:

```
for word, count in words_and_counts.most_common():
```

```
    if count > 1:
        print(word, count)
```

Lo que da los resultados esperados (a menos que se suponga que “scikit-learn” ha quedado dividido en dos palabras, en cuyo caso no los da).

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
machine 2
neural 2
scikit-learn 2
r 2
```

En el capítulo 21 veremos maneras más sofisticadas de extraer temas de datos.

Sigamos adelante

¡Ha sido un día bastante fructuoso! Cansado, sale del edificio sigilosamente, antes de que alguien pueda pedirle algo más. Descanse bien esta noche, porque mañana tendrá su sesión de orientación al empleado (sí, ha tenido un completo día de trabajo sin tan siquiera pasar por orientación al empleado; háblelo con RR. HH.).

---

[1] https://theblog.okcupid.com/the-most-important-questions-on-okcupid-32e80bad0854.

[2] https://www.facebook.com/notes/10158928002728415/.

[3] https://www.facebook.com/notes/10158927994943415/.

[4] https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html.

[5] https://www.wired.com/2016/11/facebook-won-trump-election-not-just-fake-news/.

[6] https://www.marketplace.org/2014/08/22/tech/beyond-ad-clicks-using-big-data-social-good.

[7] https://dssg.uchicago.edu/2014/08/20/tracking-the-paths-of-homelessness/.

2 Un curso acelerado de Python

La gente sigue loca por Python tras veinticinco años, cosa que me resulta difícil de creer.

—Michael Palin

Todos los empleados nuevos de DataSciencester tienen que pasar obligadamente por orientación al empleado, cuya parte más interesante es un curso acelerado de Python.

No se trata de un tutorial extenso, sino que está destinado a destacar las partes del lenguaje que serán más importantes para nosotros (algunas de las cuales no suelen ser el objetivo de los tutoriales de Python habituales). Si nunca había utilizado Python antes, probablemente quiera complementar esto con algún tipo de tutorial para principiante.

El zen de Python

Python tiene una descripción un poco zen de sus principios de diseño,[1] que se pueden encontrar dentro del propio intérprete de Python escribiendo `import this` (importar esto). Uno de los más discutidos es:

There should be one—and preferably only one—obvious way to do it.

(Solo debería haber una, y preferiblemente solo una, forma obvia de hacerlo).

El código escrito de acuerdo con esta forma "obvia" (que no tiene por qué serlo en absoluto para un principiante) se describe a menudo como "*pythonic*" (emplearemos la traducción "pitónico" en castellano, aunque suene un poco raro). Aunque este libro no trata de Python, de vez en cuando

contrastaremos formas pitónicas y no pitónicas de realizar las mismas cosas, y en general tenderemos más a emplear las soluciones pitónicas a nuestros problemas.

Otros principios aluden a lo estético:

Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex.

> *(Lo bello es mejor que lo feo. Lo explícito es mejor que lo implícito. Lo simple es mejor que lo complejo).*

Y representan ideales por los que lucharemos en nuestro código.

Conseguir Python

> **Nota:** Como las instrucciones de instalación de las cosas pueden cambiar, mientras que los libros impresos no, se pueden encontrar instrucciones actualizadas para instalar Python en el repositorio GitHub del libro.[2]
> Conviene revisar las del sitio web si las incluidas aquí no funcionan.

Se puede descargar Python desde Python.org.[3] Pero, si todavía no se dispone de él, es más recomendable instalar la distribución Anaconda,[4] que ya incluye la mayoría de las librerías necesarias para hacer ciencia de datos.

Cuando escribí la primera versión de este volumen, Python 2.7 seguía siendo el preferido por la mayoría de los científicos de datos. Por eso la primera edición del libro estaba basada en esta versión.

Pero, en los últimos años, casi todo el mundo ha migrado a Python 3. Las últimas versiones de Python tienen muchas funciones que permiten escribir código limpio con mayor facilidad, y nos aprovecharemos de otras que solo están disponibles en la versión 3.6 de Python o posterior, lo que significa que habría que conseguir esta versión más reciente de Python (además, muchas librerías útiles ya no soportan Python 2.7; otra razón más para cambiar).

Entornos virtuales

Ya desde el próximo capítulo utilizaremos la librería matplotlib para generar gráficas y diagramas o tablas. Esta librería no forma parte de Python; hay que instalarla por separado. Todos los proyectos de ciencia de datos que realicemos requerirán alguna combinación de librerías externas, a veces con versiones específicas que difieren de las empleadas en otros proyectos. Si tuviéramos una única instalación de Python, estas librerías entrarían en conflicto y provocarían todo tipo de problemas.

La solución estándar es utilizar entornos virtuales, es decir, entornos aislados de Python que mantienen sus propias versiones de librerías del lenguaje (y, dependiendo del modo en que se configure el entorno, del lenguaje en sí mismo).

Recomiendo instalar la distribución de Python denominada Anaconda, de modo que en esta sección explicaré cómo funcionan los entornos de Anaconda. También se puede utilizar el módulo venv integrado[5] o instalar virtualenv,[6] en cuyo caso habría que seguir sus instrucciones.

Para crear un entorno virtual (Anaconda) basta con hacer lo siguiente:

```
# crear un entorno Python 3.6 llamado "dsfs"
conda create -n dsfs python=3.6
```

Siguiendo los mensajes, logramos un entorno virtual llamado “dsfs”, con estas instrucciones:

```
#
# Para activar este entorno utilice:
# > source activate dsfs
#
# Para desactivar un entorno activo utilice:
# > source deactivate
#
```

Como se indica, el entorno se activa utilizando:

```
source activate dsfs
```

En ese punto, la línea de comandos debería cambiar para indicar el entorno activo. En mi MacBook aparece ahora lo siguiente en la línea de comandos:

```
(dsfs) ip-10-0-0-198:~ joelg$
```

Siempre que este entorno esté activo, las librerías se instalarán únicamente en el entorno dsfs. Cuando termine este libro y cree sus propios proyectos, debería crear sus propios entornos para ellos.

Ahora que tenemos el entorno, podemos instalar IPython,[7] que es un *shell* o intérprete de Python completo:

```
python -m pip install ipython
```

> **Nota:** Anaconda incluye su propio gestor de paquetes, `conda`, pero se puede utilizar tranquilamente el gestor de paquetes estándar de Python, `pip`, que es lo que haremos.

El resto de este libro supondrá que se ha creado y activado dicho entorno virtual de Python 3.6 (aunque le puede llamar como le parezca), y los últimos capítulos podrían hacer referencia a las librerías cuya instalación indiqué en anteriores capítulos.

Por una cuestión de disciplina, sería conveniente trabajar siempre en un entorno virtual y no utilizar nunca la instalación "básica" de Python.

Formato con espacios en blanco

Muchos lenguajes utilizan llaves para delimitar los bloques de código. Python emplea la sangría:

```
# El signo de la almohadilla marca el comienzo de un comentario. Python
# ignora los comentarios, pero son útiles para cualquiera que lea el código.
for i in [1, 2, 3, 4, 5]:
   print(i)                          # primera línea del bloque "for i"
   for j in [1, 2, 3, 4, 5]:
      print(j)                       # primera línea del bloque "for j"
      print(i + j)                   # última línea del bloque "for j"
```

```
  print(i)                          # última línea del bloque "for i"
print("done looping")
```

Así, el código de Python resulta fácilmente legible, pero también significa que hay que tener mucho cuidado con el formato.

> **Advertencia:** Los programadores suelen debatir sobre si utilizar tabuladores o espacios para la sangría. En muchos lenguajes eso no importa, pero Python considera los tabuladores y los espacios como distintos tipos de sangría y el código no se ejecutará si se mezclan los dos. Al escribir en Python siempre hay que utilizar espacios, nunca tabuladores (si se escribe código en un editor es posible configurarlo de forma que la tecla **Tab** inserte espacios).

Los espacios en blanco se ignoran dentro de paréntesis y corchetes, lo que puede resultar útil para cálculos largos:

```
long_winded_computation       (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12
=                             +
                              13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

Y para que el código resulte más fácil de leer:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
easier_to_read_list_of_lists =       [[1, 2, 3],
                                      [4, 5, 6],
                                      [7, 8, 9]]
```

También se puede utilizar una barra invertida para indicar que una sentencia continúa en la siguiente línea, aunque pocas veces se hace:

```
two_plus_three =2 + \
                3
```

Una consecuencia del formato con espacios en blanco es que puede ser difícil copiar y pegar código en el intérprete de Python. Por ejemplo, al intentar pegar este código:

```
for i in [1, 2, 3, 4, 5]:
   # observe la línea en blanco
   print(i)
```

En el *shell* normal de Python, aparecería este error:

```
IndentationError: expected an indented block
```

Porque el intérprete piensa que la línea vacía señala el final del bloque del bucle `for`.

IPython tiene una función mágica llamada `%paste` que pega correctamente lo que haya en el portapapeles, con espacios en blanco y todo. Solamente esto ya es una muy buena razón para utilizar IPython.

Módulos

Ciertas funciones de Python no se cargan por defecto. Entre ellas, hay funciones que están incluidas como parte del lenguaje y funciones externas que cada usuario puede descargar por su cuenta. Para poder utilizar estas funciones es necesario importar los módulos que las contienen.

Una forma de hacer esto es simplemente importando el propio módulo:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Aquí, `re` es el módulo que contiene funciones y constantes para trabajar con expresiones regulares. Tras este tipo de `import` hay que poner delante de esas funciones el prefijo `re.` para poder acceder a ellas.

Si ya había un `re` distinto en el código que se está utilizando, se puede emplear otro nombre:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

También se podría hacer esto si el módulo tiene un nombre poco manejable o si se va a escribir muy a menudo. Por ejemplo, un convenio estándar cuando se visualizan datos con matplotlib es:

```
import matplotlib.pyplot as plt
plt.plot(...)
```

Si se necesitan algunos valores específicos de un módulo, se pueden importar de manera explícita y utilizarlos sin reservas:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

Siendo malas personas, podríamos importar el contenido completo de un módulo en nuestro espacio de nombres, lo que podría sobrescribir involuntariamente variables que ya estaban definidas:

```
match = 10
from re import *      # oh, oh, re tiene una función que se llama igual
print(match)          # "<function match at 0x10281e6a8>"
```

Pero, como en realidad no somos malas personas, nunca haremos esto.

Funciones

Una función es una regla para tomar cero o más entradas y devolver una salida correspondiente. En Python, las funciones se definen normalmente utilizando `def`:

```
def double(x):
    """
    This is where you put an optional docstring that explains what the
    function does. For example, this function multiplies its input by 2.
    """
    return x * 2
```

Las funciones de Python son de primera clase, lo que significa que podemos asignarlas a variables y pasarlas a funciones como si se tratara de cualesquiera otros argumentos:

```
def apply_to_one(f):
```

```
  """Calls the function f with 1 as its argument"""
  return f(1)
my_double = double              # se refiere a la función anteriormente
                                definida
x =                             # es igual a 2
apply_to_one(my_double)
```

También es fácil crear funciones anónimas cortas, denominadas lambdas:

```
y = apply_to_one(lambda x: x + 4)     # es igual a 5
```

Se pueden asignar lambdas a variables, aunque la mayoría de la gente dirá que en su lugar solo hay que utilizar `def`:

```
another_double = lambda x: 2 * x      # no haga esto
def another_double(x):
  """Do this instead"""
  return 2 * x
```

A los parámetros de función también se les pueden asignar argumentos predeterminados, que solamente tienen que especificarse cuando se desea un valor distinto al predeterminado:

```
def my_print(message = "my default message"):
  print(message)
my_print("hello")     # imprime ‘hello’
my_print()            # imprime ‘my default message’
```

Algunas veces es útil especificar argumentos por el nombre:

```
def full_name(first = "What’s-his-name", last = "Something"):
  return first + " " + last
full_name("Joel", "Grus")          # "Joel Grus"
full_name("Joel")                  # "Joel Something"
full_name(last="Grus")             # "What’s-his-name Grus"
```

Vamos a crear muchas muchas funciones.

Cadenas

Las cadenas (o *strings*) pueden estar delimitadas por comillas simples o dobles (pero las comillas tienen que ir en pares):

```
single_quoted_string = ‘data science’
double_quoted_string = "data science"
```

Python utiliza las barras invertidas para codificar caracteres especiales. Por ejemplo:

```
tab_string = "\t"      # representa el carácter del tabulador
len(tab_string)        # es 1
```

Si queremos barras invertidas como tales (las que se utilizan en nombres de directorio de Windows o en expresiones regulares), se pueden crear cadenas en bruto (*raw strings*) utilizando `r""`:

```
not_tab_string = r"\t"      # representa los caracteres ‘\’ y ‘t’
len(not_tab_string)         # es 2
```

Se pueden crear cadenas multilínea utilizando comillas triples:

```
multi_line_string = """This is the first line,
and this is the second line
and this is the third line."""
```

Una función nueva en Pyhton es la *f-string*, que ofrece una sencilla manera de sustituir valores por cadenas. Por ejemplo, si nos dieran el nombre y el apellido por separado:

```
first_name = "Joel"
last_name = "Grus"
```

Querríamos combinarlos como un nombre completo. Hay distintas formas de construir una cadena `full_name`:

```
full_name1 = first_name + " " + last_name                 # suma de cadenas
full_name2 = "{0} {1}".format(first_name, last_name)      # string.format
```

Pero el método f-string es mucho más manejable:

```
full_name3 = f”{first_name} {last_name}”
```

Y lo preferiremos a lo largo del libro.

Excepciones

Cuando algo va mal, Python levanta una excepción. Si no se controlan adecuadamente, las excepciones pueden hacer que el programa se cuelgue. Se pueden manejar utilizando `try` y `except`:

```
try:
   print(0 / 0)
except ZeroDivisionError:
   print("cannot divide by zero")
```

Aunque en muchos lenguajes las excepciones no están bien consideradas, en Python no hay problema en utilizarlas para que el código sea más limpio, así que en ocasiones lo haremos.

Listas

Probablemente la estructura de datos más esencial de Python es la lista, que no es más que una colección ordenada (similar a lo que en otros lenguajes se podría denominar *array*, pero con funcionalidad añadida):

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [integer_list, heterogeneous_list, []]
list_length = len(integer_list)        # es igual a 3
list_sum = sum(integer_list)           # es igual a 6
```

Se puede obtener o establecer el elemento *n* de una lista con corchetes:

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
zero = x[0]       # es igual a 0, las listas están indexadas al 0
```

```
one = x[1]         # es igual a 1
nine = x[-1]       # es igual a 9, 'pitónico' para el último elemento
eight = x[-2]      # es igual a 8, 'pitónico' para el penúltimo elemento
x[0] = -1          # ahora x es [-1, 1, 2, 3, ..., 9]
```

También se pueden utilizar corchetes para crear *slices* en listas (cortes o arreglos). El corte `i:j` significa todos los elementos desde `i` (incluido) hasta `j` (excluido). Si dejamos fuera el principio del corte, lo extraeremos desde el principio de la lista, pero, si dejamos fuera el final del corte, lo extraeremos hasta el final:

```
first_three = x[:3]                  # [-1, 1, 2]
three_to_end = x[3:]                 # [3, 4, ..., 9]
one_to_four = x[1:5]                 # [1, 2, 3, 4]
last_three = x[-3:]                  # [7, 8, 9]
without_first_and_last = x[1:-1]     # [1, 2, ..., 8]
copy_of_x = x[:]                     # [-1, 1, 2, ..., 9]
```

De forma similar se pueden crear cortes de cadenas y otros tipos "secuenciales".

Un corte puede admitir un tercer argumento para indicar su *stride* (avance), que puede ser negativo:

```
every_third = x[::3]          # [-1, 3, 6, 9]
five_to_three = x[5:2:-1]     # [5, 4, 3]
```

Python tiene un operador `in` para comprobar los miembros de la lista:

```
1 in [1, 2, 3]     # True
0 in [1, 2, 3]     # False
```

Esta comprobación implica examinar los elementos de la lista de uno en uno, lo que significa que probablemente no se debería utilizar a menos que se sepa que la lista es pequeña (o a menos que no nos preocupe el tiempo que tarde en hacerse la comprobación).

Es fácil concatenar listas. Si se desea modificar una lista en su lugar, se puede utilizar `extend` para añadir elementos de otra colección:

```
x = [1, 2, 3]
x.extend([4, 5, 6])     # x es ahora [1, 2, 3, 4, 5, 6]
```

Si no queremos modificar `x`, podemos ampliar la lista:

```
x = [1, 2, 3]
y = x + [4, 5, 6]     # y es [1, 2, 3, 4, 5, 6]; x no ha cambiado
```

Lo más frecuente que haremos será añadir elementos a listas uno a uno:

```
x = [1, 2, 3]
x.append(0)     # x es ahora [1, 2, 3, 0]
y = x[-1]       # es igual a 0
z = len(x)      # es igual a 4
```

A menudo, es conveniente desempaquetar listas cuando se sabe cuántos elementos contienen:

```
x, y = [1, 2]     # ahora x es 1, y es 2
```

Aunque obtendremos un `ValueError` si no tenemos el mismo número de elementos en ambos lados.

Algo que se utiliza habitualmente es un carácter de subrayado para un valor del que nos vamos a deshacer:

```
_, y = [1, 2]     # ahora y == 2, no nos importa el primer elemento
```

Tuplas

Las tuplas son las primas inmutables de las listas. Casi todo lo que se le puede hacer a una lista que implique modificarla, se le puede hacer a una tupla. Se especifica una tupla utilizando paréntesis (o nada) en lugar de corchetes:

```
my_list = [1, 2]
my_tuple = (1, 2)
```

```
other_tuple = 3, 4
my_list[1] = 3      # my_list es ahora[1, 3]
try:
my_tuple[1] = 3
except TypeError:
print("cannot modify a tuple")
```

Las tuplas son una forma cómoda de devolver varios valores de funciones:

```
def sum_and_product(x, y):
   return (x + y), (x * y)
sp = sum_and_product(2, 3)         # sp es (5, 6)
s, p = sum_and_product(5, 10)      # s es 15, p es 50
```

Las tuplas (y las listas) se pueden utilizar para asignación múltiple:

```
x, y = 1,        # ahora x es 1, y es 2
2
x, y = y,        # Forma pitónica de intercambiar variables; ahora x es 2, y es
x                1
```

Diccionarios

Otra estructura de datos fundamental es un diccionario, que asocia valores a claves y permite recuperar rápidamente el valor correspondiente a una determinada clave:

```
empty_dict = {}                       # pitónico
empty_dict2 = dict()                  # menos pitónico
grades = {"Joel": 80, "Tim": 95}      # dict literal
```

Se puede consultar el valor para una clave utilizando corchetes:

```
  joels_grade = grades[“Joel”]      # es igual a 80
```

Pero se obtendrá un `KeyError` si se pregunta por una clave que no está en el diccionario:

```
try:
   kates_grade = grades["Kate"]
except KeyError:
   print("no grade for Kate!")
```

Se puede comprobar la existencia de una clave utilizando `in`:

```
joel_has_grade = "Joel" in grades      # True
kate_has_grade = "Kate" in grades      # False
```

Esta verificación de membresía es aún más rápida para diccionarios grandes.

Los diccionarios tienen un método `get` que devuelve un valor predeterminado (en lugar de levantar una excepción) cuando se consulta una clave que no está en el diccionario:

```
joels_grade = grades.get("Joel", 0)       # es igual a 80
kates_grade = grades.get("Kate", 0)       # es igual a 0
no_ones_grade = grades.get("No One")      # el valor predeterminado es None
```

Se pueden asignar pares clave/valor utilizando los mismos corchetes:

```
grades["Tim"] = 99              # reemplaza el valor anterior
grades["Kate"] = 100            # añade una tercera entrada
num_students = len(grades)      # es igual a 3
```

Como vimos en el capítulo 1, se pueden utilizar diccionarios para representar datos estructurados:

```
tweet = {
   "user" : "joelgrus",
   "text" : "Data Science is Awesome",
   "retweet_count" : 100,
   "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Aunque pronto veremos un enfoque mejor.

Además de buscar claves específicas, también podemos mirarlas todas:

```
tweet_keys = tweet.keys()       # iterable para las claves
tweet_values =                  # iterable para los valores
tweet.values()
tweet_items =                   # iterable para las tuplas (clave, valor)
tweet.items()
"user" in tweet_keys            # True, pero no pitónico
"user" in tweet                 # forma pitónica de comprobar claves
"joelgrus" in tweet_values      # True (es lenta, pero la única forma de
                                verificar)
```

Las claves de diccionario pueden ser "*hashables*"; en particular, no se pueden utilizar listas como claves. Si se necesita una clave multiparte, probablemente se debería utilizar una tupla o idear un modo de convertir la clave en una cadena.

defaultdict

Imaginemos que estamos intentando contar las palabras de un documento. Un método obvio para lograrlo es crear un diccionario en el que las claves sean palabras y los valores sean contadores. Al comprobar cada palabra, se puede incrementar su contador si ya está en el diccionario y añadirlo al diccionario si no estaba:

```
word_counts = {}
for word in document:
   if word in word_counts:
      word_counts[word] += 1
   else:
      word_counts[word] = 1
```

Se podría utilizar también el sistema "mejor pedir perdón que permiso" y simplemente manejar la excepción al intentar consultar una clave inexistente:

```
word_counts = {}
for word in document:
   try:
      word_counts[word] += 1
   except KeyError:
      word_counts[word] = 1
```

Un tercer enfoque es utilizar `get`, que se comporta con mucha elegancia con las claves inexistentes:

```
word_counts = {}
for word in document:
   previous_count = word_counts.get(word, 0)
   word_counts[word] = previous_count + 1
```

Todo esto es muy poco manejable, razón por la cual `defaultdict` es útil. Un `defaultdict` es como un diccionario normal, excepto que, cuando se intenta buscar una clave que no contiene, primero añade un valor para ella utilizando una función de argumento cero suministrada al crearla. Para utilizar diccionarios `defaultdicts`, es necesario importarlos de `collections`:

```
from collections import defaultdict
word_counts = defaultdict(int)     # int() produce 0
for word in document:
   word_counts[word] += 1
```

También pueden resultar útiles con `list` o `dict`, o incluso con nuestras propias funciones:

```
dd_list = defaultdict(list)              # list() produce una lista vacía
dd_list[2].append(1)                     # ahora dd_list contiene {2: [1]}
dd_dict = defaultdict(dict)              # dict() produce un dict vacío
dd_dict["Joel"]["City"] = "Seattle"      # {"Joel" : {"City": Seattle"}}
dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                        # ahora dd_pair contiene {2: [0, 1]}
```

Serán útiles cuando estemos utilizando diccionarios para “recopilar” resultados según alguna clave y no queramos comprobar todo el tiempo si la clave sigue existiendo.

Contadores

Un `Counter` convierte una secuencia de valores en un objeto de tipo

`defaultdict(int)` mapeando claves en contadores:

```
from collections import Counter
c = Counter([0, 1, 2, 0])     # c es (básicamente) {0: 2, 1: 1, 2: 1}
```

Lo que nos ofrece un modo muy sencillo de resolver problemas de `word_counts`:

```
# recuerde, document es una lista de palabras
word_counts = Counter(document)
```

Una instancia `Counter` tiene un método `most_common` que se utiliza con frecuencia:

```
# imprime las 10 palabras más comunes y sus contadores
for word, count in word_counts.most_common(10):
   print(word, count)
```

Conjuntos

Otra estructura de datos útil es el conjunto o *set*, que representa una colección de distintos elementos. Se pueden definir un conjunto listando sus elementos entre llaves:

```
primes_below_10 = {2, 3, 5, 7}
```

Sin embargo, esto no funciona con conjuntos vacíos, dado que `{}` ya significa "`dict` vacío". En ese caso, habrá que utilizar la propia `set()`:

```
s = set()
s.add(1)       # s es ahora {1}
s.add(2)       # s es ahora {1, 2}
s.add(2)       # s sigue siendo {1, 2}
x = len(s)     # es igual a 2
y = 2 in s     # es igual a True
z = 3 in s     # es igual a False
```

Utilizaremos conjuntos por dos razones principales. La primera es que `in`

es una operación muy rápida con conjuntos. Si tenemos una gran colección de elementos que queremos utilizar para hacer una prueba de membresía, un conjunto es más adecuado que una lista:

```
stopwords_list = ["a", "an", "at"] + hundreds_of_other_words + ["yet", "you"]
"zip" in stopwords_list       # False, pero hay que verificar cada elemento
stopwords_set = set(stopwords_list)
"zip" in stopwords_set        # muy rápido de comprobar
```

La segunda razón es encontrar los elementos distintos de una colección:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)              # 6
item_set = set(item_list)               # {1, 2, 3}
num_distinct_items = len(item_set)      # 3
distinct_item_list = list(item_set)     # [1, 2, 3]
```

Utilizaremos conjuntos con menos frecuencia que diccionarios y listas.w

Flujo de control

Como en la mayoría de los lenguajes de programación, se puede realizar una acción de forma condicional utilizando `if`:

```
if 1 > 2:
   message = "if only 1 were greater than two..."
elif 1 > 3:
   message = "elif stands for ‘else if’"
else:
   message = "when all else fails use else (if you want to)"
```

Se puede también escribir un ternario if-then-else en una sola línea, cosa que haremos muy de tanto en tanto:

```
parity = “even” if x % 2 == 0 else “odd”
```

Python tiene un bucle `while`:

```
x = 0
while x < 10:
   print(f"{x} is less than 10")
   x += 1
```

Aunque con mucha más frecuencia usaremos `for` e `in`:

```
# range(10) es los números 0, 1, ..., 9
for x in range(10):
   print(f"{x} is less than 10")
```

Si necesitáramos lógica más compleja, podríamos utilizar `continue` y `break`:

```
for x in range(10):
   if x == 3:
      continue      # va inmediatamente a la siguiente repetición
   if x == 5:
      break         # sale del todo del bucle
   print(x)
```

Esto imprimirá 0, 1, 2 y 4.

Verdadero o falso

Los valores booleanos en Python funcionan igual que en casi todos los demás lenguajes, excepto que llevan la primera letra en mayúscula:

```
one_is_less_than_two = 1 < 2          # es igual a True
true_equals_false = True == False     # es igual a False
```

Python utiliza el valor `None` para indicar un valor no existente. Es similar al `null` de otros lenguajes:

```
x = None
assert x == None, "this is the not the Pythonic way to check for None"
assert x is None, "this is the Pythonic way to check for None"
```

Python permite utilizar cualquier valor donde espera un booleano. Las siguientes expresiones son todas “falsas”:

- `False`.
- `None`.
- `[]` (una `list` vacía).
- `{}` (un `dict` vacío).
- `""`.
- `set()`.
- `0`.
- `0.0`.

Casi todo lo demás se trata como `True`. Ello permite utilizar fácilmente sentencias `if` para probar listas vacías, cadenas vacías, diccionarios vacíos, etc. También produce en ocasiones errores complicados si no se espera este comportamiento:

```
s = some_function_that_returns_a_string()
if s:
   first_char = s[0]
else:
   first_char = ""
```

Una forma más corta (pero posiblemente más confusa) de hacer lo mismo es:

```
first_char = s and s[0]
```

Ya que `and` devuelve su segundo valor cuando el primero es “verdadero”, y el primer valor cuando no lo es. De forma similar, si `x` es o bien un número o posiblemente `None`:

```
safe_x = x or 0
```

Es definitivamente un número, aunque:

```
safe_x = x if x is not None else 0
```

Es posiblemente más legible.

Python tiene una función `all`, que toma un iterable y devuelve `True` exactamente cuando cada elemento es verdadero, y una función `any`, que devuelve `True` cuando al menos un elemento es verdad:

```
all([True, 1, {3}])     # True, todos son verdaderos
all([True, 1, {}])      # False, {} is falso
any([True, 1, {}])      # True, True is verdadero
all([])                 # True, no hay elementos falsos en la lista
any([])                 # False, no hay elementos verdaderos en la lista
```

Ordenar

Toda lista de Python tiene un método `sort` que la ordena en su lugar. Si no queremos estropear nuestra lista, podemos usar la función `sorted`, que devuelve una lista nueva:

```
x = [4, 1, 2, 3]
y = sorted(x)     # y es [1, 2, 3, 4], x queda igual
x.sort()          # ahora x es [1, 2, 3, 4]
```

Por defecto, `sort` (y `sorted`) ordena una lista de menor a mayor basándose en comparar inocentemente los elementos uno con otro.

Si queremos que los elementos estén ordenados de mayor a menor, se puede especificar un parámetro `reverse=True`. Y, en lugar de comparar los elementos por sí mismos, se pueden comparar los resultados de una función que se especifica con `key`:

```
# ordena la lista por valor absoluto de mayor a menor
x = sorted([-4, 1, -2, 3], key=abs, reverse=True)         # is [-4, 3, -2, 1]
# ordena las palabras y contadores del contador mayor al menor
wc = sorted(word_counts.items(),
        key=lambda word_and_count: word_and_count[1],
        reverse=True)
```

Comprensiones de listas

Con frecuencia, vamos a querer transformar una lista en otra distinta seleccionando solo determinados elementos, transformando elementos o haciendo ambas cosas. La forma pitónica de hacer esto es con *list comprehensions*, o comprensiones de listas:

```
even_numbers = [x for x in range(5) if x % 2 == 0]     # [0, 2, 4]
squares = [x * x for x in range(5)]                    # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]           # [0, 4, 16]
```

De forma similar, se pueden convertir listas en diccionarios o conjuntos:

```
square_dict = {x: x * x for x in                # {0: 0, 1: 1, 2: 4, 3: 9, 4:
range(5)}                                       16}
square_set = {x * x for x in [1, -1]}           # {1}
```

Si no necesitamos el valor de la lista, es habitual utilizar un guion bajo como variable:

```
zeros = [0 for _ in                     # tiene la misma longitud que
even_numbers]                           even_numbers
```

Una comprensión de lista puede incluir varios `for`:

```
pairs = [(x, y)
     for x in range(10)
     for y in range(10)]     # 100 pares (0,0) (0,1) ... (9,8), (9,9)
```

Y después los `for` pueden utilizar los resultados de los anteriores:

```
increasing_pairs = [(x, y)                # solo pares con x < y,
          for x in range(10)              # range(bajo, alto) es igual a
          for y in range(x + 1, 10)]      # [bajo, bajo + 1, ..., alto-1]
```

Utilizaremos mucho las comprensiones de listas.

Pruebas automatizadas y assert

Como científicos de datos, escribiremos mucho código. ¿Cómo podemos estar seguros de que nuestro código es correcto? Una forma es con tipos (de los que hablaremos en breve), pero otra forma es con *automated tests* o pruebas automatizadas.

Hay estructuras muy complicadas para escribir y ejecutar pruebas, pero en este libro nos limitaremos a utilizar sentencias `assert`, que harán que el código levante un `AssertionError` si la condición especificada no es verdadera:

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 should equal 2 but didn't"
```

Como podemos ver en el segundo caso, se puede añadir si se desea un mensaje que se imprimirá si la evaluación falla.

No es especialmente interesante evaluar que 1 + 1 = 2, pero lo es mucho más verificar que las funciones que escribamos hagan lo que se espera de ellas:

```
def smallest_item(xs):
   return min(xs)
assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

A lo largo del libro utilizaremos `assert` de esta forma. Es una buena práctica, y yo animo a utilizar libremente esta sentencia (en el código del libro que encontramos en GitHub se comprueba que contiene muchas más sentencias `assert` de las que aparecen impresas en el libro, lo que me permite estar seguro de que el código que he escrito es correcto).

Otro uso menos habitual es evaluar cosas sobre entradas a funciones:

```
def smallest_item(xs):
   assert xs, "empty list has no smallest item"
   return min(xs)
```

Haremos esto de vez en cuando, pero será más frecuente utilizar `assert`

para verificar que el código escrito es correcto.

Programación orientada a objetos

Como muchos lenguajes, Python permite definir clases que encapsulan los datos y las funciones que operan con ellos. Las utilizaremos algunas veces para que nuestro código sea más limpio y sencillo. Probablemente, es más fácil explicarlas construyendo un ejemplo con muchas anotaciones.

Aquí vamos a crear una clase que represente un "contador de clics", del tipo de los que se ponen en la puerta para controlar cuántas personas han acudido al encuentro "Temas avanzados sobre ciencia de datos".

Mantiene un contador (`count`), se le puede hacer clic (`clicked`) para aumentar la cuenta, permite lectura de contador (`read_count`) y se puede reiniciar (`reset`) de vuelta a cero (en la vida real una de estas clases pasa de 9999 a 0000, pero no vamos a preocuparnos de eso ahora).

Para definir una clase, utilizamos la palabra clave `class` y un nombre de tipo PascalCase:

```
class CountingClicker:
  """A class can/should have a doctring, just like a function"""
```

Una clase contiene cero o más funciones miembro. Por convenio, cada una toma un primer parámetro, `self`, que se refiere a la instancia en particular de la clase.

Normalmente, una clase tiene un constructor, llamado `__init__`, que toma los parámetros que necesita para construir una instancia de dicha clase y hace cualquier configuración que se necesite:

```
def __init__(self, count = 0):
  self.count = count
```

Aunque el constructor tiene un nombre divertido, construimos las instancias del contador de clics utilizando solamente el nombre de la clase:

```
clicker1 = CountingClicker()              # inicializado a 0
clicker2 = CountingClicker(100)           # empieza con count=100
clicker3 =                                # forma más explícita de hacer lo
CountingClicker(count=100)                mismo
```

Vemos que el nombre del método `__init__` empieza y termina con guiones bajos. A veces a estos métodos "mágicos" se les llama métodos "dunder" (término inventado que viene de *doubleUNDERscore*, es decir, doble guion bajo) y representan comportamientos "especiales".

> **Nota:** Los métodos de clase cuyos nombres empiezan con un guion bajo se consideran (por convenio) "privados", y se supone que los usuarios de esa clase no les llaman directamente. Sin embargo, Python no impide a los usuarios llamarlos.

Otro método similar es `__repr__`, que produce la representación de cadena de una instancia de clase:

```
def __repr__(self):
   return f"CountingClicker(count={self.count})"
```

Y finalmente tenemos que implementar la API pública de la clase que hemos creado:

```
def click(self, num_times = 1):
   """Click the clicker some number ot times."""
   self.count += num_times
def read(self):
   return self.count
def reset(self):
   self.count = 0
```

Una vez definido, utilicemos `assert` para escribir algunos casos de prueba para nuestro contador de clics:

```
clicker = CountingClicker()
assert clicker.read() == 0, "clicker should start with count 0"
clicker.click()
```

```
clicker.click()
assert clicker.read() == 2, "after two clicks, clicker should have count 2"
clicker.reset()
assert clicker.read() == 0, "after reset, clicker should be back to 0"
```

Escribir pruebas como estas nos permite estar seguros de que nuestro código esté funcionando tal y como está diseñado, y que esto va a seguir siendo así siempre que le hagamos cambios.

También crearemos de vez en cuando subclases que heredan parte de su funcionalidad de una clase padre. Por ejemplo, podríamos crear un contador de clics no reiniciable utilizando `CountingClicker` como clase base y anulando el método `reset` para que no haga nada:

```
# Una subclase hereda todo el comportamiento de su clase padre.
class NoResetClicker(CountingClicker):
   # Esta clase tiene los mismos métodos que CountingClicker
   # Salvo que tiene un método reset que no hace nada.
   def reset(self):
      pass
clicker2 = NoResetClicker()
assert clicker2.read() == 0
clicker2.click()
assert clicker2.read() == 1
clicker2.reset()
assert clicker2.read() == 1, "reset shouldn’t do anything"
```

Iterables y generadores

Una cosa buena de las listas es que se pueden recuperar determinados elementos por sus índices. Pero ¡esto no siempre es necesario! Una lista de mil millones de números ocupa mucha memoria. Si solo queremos los elementos uno cada vez, no hay una buena razón que nos haga conservarlos a todos. Si solamente terminamos necesitando los primeros elementos, generar los mil millones es algo tremendamente inútil.

A menudo, todo lo que necesitamos es pasar varias veces por la colección utilizando `for` e `in`. En este caso podemos crear generadores, que se pueden

iterar igual que si fueran listas, pero generan sus valores bajo petición.

Una forma de crear generadores es con funciones y con el operador `yield`:

```
def generate_range(n):
   i = 0
   while i < n:
      yield I      # cada llamada a yield produce un valor del generador
      i += 1
```

El siguiente bucle consumirá uno a uno los valores a los que se ha aplicado `yield` hasta que no quede ninguno:

```
for i in generate_range(10):
   print(f"i: {i}")
```

(En realidad, `range` es bastante perezosa de por sí, así que hacer esto no tiene ningún sentido).

Con un generador, incluso se puede crear una secuencia infinita:

```
def natural_numbers():
   """returns 1, 2, 3, ..."""
   n = 1
   while True:
      yield n
      n += 1
```

Aunque probablemente no deberíamos iterar sobre él sin utilizar algún tipo de lógica de interrupción.

Truco: La otra cara de la pereza es que solo se puede iterar una única vez por un generador. Si hace falta pasar varias veces, habrá que volver a crear el generador cada vez o bien utilizar una lista. Si generar los valores resulta caro, podría ser una buena razón para utilizar una lista en su lugar.

Una segunda manera de crear generadores es utilizar las comprensiones envueltas en paréntesis:

```
evens_below_20 = (i for i in generate_range(20) if i % 2 == 0)
```

Una “comprensión de generador” como esta no hace nada hasta que se itera sobre ella (utilizando `for` o `next`). Podemos utilizar esto para crear complicadas líneas de proceso de datos:

```
# Ninguno de estos cálculos *hace* nada hasta que iteramos
data = natural_numbers()
evens = (x for x in data if x % 2 == 0)
even_squares = (x ** 2 for x in evens)
even_squares_ending_in_six = (x for x in even_squares if x % 10 == 6)
# y así sucesivamente
```

No pocas veces, cuando estemos iterando sobre una lista o un generador, no querremos solamente los valores, sino también sus índices. Para este caso habitual, Python ofrece una función `enumerate`, que convierte valores en pares (`index`, `value`):

```
names = ["Alice", "Bob", "Charlie", "Debbie"]
# no pitónico
for i in range(len(names)):
   print(f"name {i} is {names[i]}")
# tampoco pitónico
i = 0
for name in names:
   print(f"name {i} is {names[i]}")
   i += 1
# pitónico
for i, name in enumerate(names):
   print(f"name {i} is {name}")
```

Utilizaremos mucho esto.

Aleatoriedad

A medida que aprendamos ciencia de datos, necesitaremos con frecuencia generar números aleatorios, lo que podemos hacer con el módulo `random`:

```
import random
random.seed(10)     # esto asegura que obtenemos los mismos resultados cada vez
```

```
four_uniform_randoms = [random.random() for _ in range(4)]

# [0.5714025946899135,      # random.random() produce números
# 0.4288890546751146,       # de manera uniforme entre 0 y 1.
# 0.5780913011344704,       # Es la función random que utilizaremos
# 0.20609823213950174]      # con más frecuencia.
```

El módulo `random` produce en realidad números pseudoaleatorios (es decir, deterministas) basados en un estado interno que se puede configurar con `random.seed` si lo que se desea es obtener resultados reproducibles:

```
random.seed(10)             # establece la semilla en 10
print(random.random())      # 0.57140259469
random.seed(10)             # reinicia la semilla en 10
print(random.random())      # 0.57140259469 de nuevo
```

Algunas veces utilizaremos `random.randrange`, que toma uno o dos argumentos y devuelve un elemento elegido aleatoriamente del `range` correspondiente:

```
random.randrange(10)      # selecciona aleatoriamente de range(10) = [0, 1,
                          ..., 9]
random.randrange(3,       # selecciona aleatoriamente de range(3, 6) = [3, 4,
6)                        5]
```

Hay varios métodos más que en ocasiones nos resultarán convenientes. Por ejemplo, `random.shuffle` reordena aleatoriamente los elementos de una lista:

```
up_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(up_to_ten)
print(up_to_ten)
# [7, 2, 6, 8, 9, 4, 10, 1, 3,      (sus resultados serán probablemente
5]                                  diferentes)
```

Si se necesita elegir aleatoriamente un elemento de una lista, se puede utilizar `random.choice`:

```
my_best_friend = random.choice([“Alice”, “Bob”, “Charlie”])      # “Bob” para mí
```

Y, si lo que hace falta es elegir aleatoriamente una muestra de elementos sin sustituto (es decir, sin duplicados), se puede utilizar `random.sample`:

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers,        # [16, 36, 10, 6, 25,
6)                                                      9]
```

Para elegir una muestra de elementos con sustituto (es decir, que permita duplicados), simplemente basta con hacer varias llamadas a `random.choice`:

```
four_with_replacement = [random.choice(range(10)) for _ in range(4)]
print(four_with_replacement)                  # [9, 4, 4, 2]
```

Expresiones regulares

Las expresiones regulares ofrecen un modo de buscar texto. Son increíblemente útiles, pero también bastante complicadas (tanto, que hay escritos libros enteros sobre ellas). Entraremos en detalle las pocas veces que nos las encontremos; estos son algunos ejemplos de cómo utilizarlas en Python:

```
import re
re_examples = [                                 # Todas son True, porque
   not re.match("a", "cat"),                    # 'cat' no empieza por 'a'
   re.search("a", "cat"),                       # 'cat' contiene una 'a'
   not re.search("c", "dog"),                   # 'dog' no contiene una 'c'.
   3 == len(re.split("[ab]",                    # Partido en a o b para
   "carbs")),                                   ['c','r','s'].
   "R-D-" == re.sub("[0-9]", "-",               # Reemplaza dígitos por guiones.
   "R2D2")
   ]
assert all(re_examples), "all the regex examples should be True"
```

Algo importante a tener en cuenta es que `re.match` comprueba si el principio de una cadena coincide con una expresión regular, mientras que `re.search` lo comprueba con alguna parte de una cadena. En algún momento es probable que se mezclen los dos y creen problemas.

La documentación oficial en https://docs.python.org/es/3/library/re.html ofrece muchos más detalles.

Programación funcional

> **Nota:** La primera edición de este libro presentaba en este momento las funciones de Python `partial`, `map`, `reduce` y `filter`. En mi viaje hacia la iluminación, me he dado cuenta de que es mejor evitarlas, y sus usos en el libro han sido reemplazados por comprensiones de listas, bucles `for` y otras construcciones más pitónicas.

Empaquetado y desempaquetado de argumentos

A menudo, necesitaremos empaquetar (*zip*) dos o más iterables juntos. La función `zip` transforma varios iterables en uno solo de tuplas de función correspondiente:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
# zip es perezoso, de modo que hay que hacer algo como lo siguiente
[pair for pair in zip(list1, list2)]     # es [('a', 1), ('b', 2), ('c', 3)]
```

Si las listas tienen distintas longitudes, `zip` se detiene tan pronto como termina la primera lista.

También se puede "desempaquetar" una lista utilizando un extraño truco:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

El asterisco (*) realiza desempaquetado de argumento, que utiliza los elementos de `pairs` como argumentos individuales para `zip`. Termina igual que si lo hubiéramos llamado:

```
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
```

Se puede utilizar desempaquetado de argumento con cualquier función:

```
def add(a, b): return a + b
add(1, 2)          # devuelve 3
try:
   add([1, 2])
except TypeError:
   print("add expects two inputs")
add(*[1, 2])       # devuelve 3
```

Es raro que encontremos esto útil, pero, cuando lo hacemos, es un truco genial.

args y kwargs

Digamos que queremos crear una función de máximo orden que requiere como entrada una función `f` y devuelve una función nueva que para cualquier entrada devuelve el doble del valor de `f`:

```
def doubler(f):
   # Aquí definimos una nueva función que mantiene una referencia a f
   def g(x):
      return 2 * f(x)
   # Y devuelve esa nueva función
   return g
```

Esto funciona en algunos casos:

```
def f1(x):
   return x + 1
g = doubler(f1)
assert g(3) == 8, "(3 + 1) * 2 should equal 8"
assert g(-1) == 0, "(-1 + 1) * 2 should equal 0"
```

Sin embargo, no sirve con funciones que requieren algo más que un solo argumento:

```
def f2(x, y):
```

```
   return x + y
g = doubler(f2)
try:
   g(1, 2)
except TypeError:
   print("cas defined, g only takes one argument")
```

Lo que necesitamos es una forma de especificar una función que tome argumentos arbitrarios. Podemos hacerlo con desempaquetado de argumento y un poco de magia:

```
def magic(*args, **kwargs):
   print("unnamed args:", args)
   print("keyword args:", kwargs)
magic(1, 2, key="word", key2="word2")
# imprime
# argumentos sin nombre: (1, 2)
# argumentos de palabra clave: {‘key’: ‘word’, ‘key2’: ‘word2’}
```

Es decir, cuando definimos una función como esta, `args` es una tupla de sus argumentos sin nombre y `kwargs` es un `dict` de sus argumentos con nombre. Funciona también a la inversa, si queremos utilizar una `list` (o `tuple`) y `dict` para proporcionar argumentos a una función:

```
def other_way_magic(x, y, z):
   return x + y + z
x_y_list = [1, 2]
z_dict = {"z": 3}
assert other_way_magic(*x_y_list, **z_dict) == 6, "1 + 2 + 3 should be 6"
```

Se podría hacer todo tipo de trucos extraños con esto; solo lo utilizaremos para producir funciones de máximo orden cuyas entradas puedan aceptar argumentos arbitrarios:

```
def doubler_correct(f):
   """works no matter what kind of inputs f expects"""
   def g(*args, **kwargs):
      """whatever arguments g is supplied, pass them through to f"""
      return 2 * f(*args, **kwargs)
   return g
```

```
g = doubler_correct(f2)
assert g(1, 2) == 6, "doubler should work now"
```

Como regla general, el código que escribamos será más correcto y legible si somos explícitos en lo que se refiere a los tipos de argumentos que las funciones que usemos requieren; de ahí que vayamos a utilizar `args` y `kwargs` solo cuando no tengamos otra opción.

Anotaciones de tipos

Python es un lenguaje de tipos dinámicos. Esto significa que en general no le importan los tipos de objetos que utilicemos, siempre que lo hagamos de formas válidas:

```
def add(a, b):
   return a + b
assert add(10, 5) == 15, "+ is valid for numbers"
assert add([1, 2], [3]) == [1, 2, 3], "+ is valid for lists"
assert add("hi ", "there") == "hi there", "+ is valid for strings"
try:
   add(10, "five")
except TypeError:
   print("cannot add an int to a string")
```

Mientras que en un lenguaje de tipos estáticos nuestras funciones y objetos tendrían tipos específicos:

```
def add(a: int, b: int) -> int:
   return a + b
add(10, 5)              # le gustaría que esto fuera correcto
add("hi ", "there")     # le gustaría que esto no fuera correcto
```

En realidad, las versiones más recientes de Python tienen (más o menos) esta funcionalidad. ¡La versión anterior de `add` con las anotaciones de tipos `int` es válida en Python 3.6! Sin embargo, estas anotaciones de tipos no hacen realmente nada. Aún se puede utilizar la función anotada `add` para

añadir cadenas, y la llamada a `add(10, “five”)` seguirá levantando exactamente el mismo `TypeError`.

Dicho esto, sigue habiendo (al menos) cuatro buenas razones para utilizar anotaciones de tipos en el código Python que escribamos:

- Los tipos son una forma importante de documentación. Esto es doblemente cierto en un libro que utiliza código para enseñar conceptos teóricos y matemáticos. Comparemos las siguientes dos líneas de función:

```
def dot_product(x, y): ...
# aún no hemos definido Vector, pero imagínese que lo habíamos hecho
def dot_product(x: Vector, y: Vector) -> float: ...
```

  Encuentro el segundo extremadamente más informativo; espero que también se lo parezca (en este punto me he acostumbrado tanto a la determinación de tipos que ahora sin ello encuentro Python difícil de leer).
- Hay herramientas externas (siendo la más popular `mypy`) que leerán el código que escribamos, inspeccionarán las anotaciones de tipos y ofrecerán errores de tipos antes siquiera de ejecutar el código. Por ejemplo, si ejecutamos `mypy` en un archivo que contiene `add(“hi “,”there”)`, avisaría de lo siguiente:

```
error: Argument 1 to “add” has incompatible type “str”; expected “int”
```

  Al igual que la prueba `assert`, esta es una buena forma de encontrar errores en el código antes de ejecutarlo. La narración del libro no implicará tal comprobación de tipo; sin embargo, ejecutaré una en segundo plano, lo que me permitirá asegurarme de que el libro en sí es correcto.
- Tener que pensar en los tipos de nuestro código nos obliga a diseñar funciones e interfaces más limpios:

```
from typing import Union
def secretly_ugly_function(value, operation): ...
```

```
def ugly_function(value: int,
                  operation: Union[str, int, float, bool]) -> int:
    ...
```

Aquí tenemos una función cuyo parámetro de operación puede ser un `string`, un `int`, un `float` o un `bool`. Es muy probable que esta función sea frágil y difícil de utilizar, pero aún queda más claro cuando los tipos resultan explícitos. Hacer esto nos obligará a diseñar de un modo menos torpe, cosa que nuestros usuarios nos agradecerán.

- Utilizar tipos permite al editor que utilicemos ayudarnos con cosas como autocompletar (véase la figura 2.1) y enfadarnos por los errores de escritura.

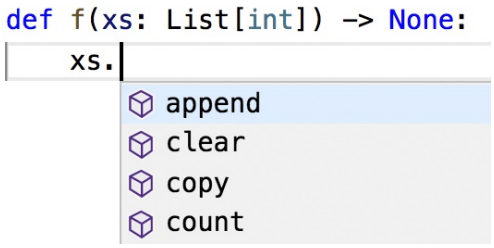


Figura 2.1. VSCode, pero probablemente otro editor haga lo mismo.

A veces, la gente insiste en que las comprobaciones de tipo pueden ser valiosas en proyectos grandes, pero no merecen la pena en otros más pequeños. No obstante, como casi no se tardan nada en escribir y permiten al editor ahorrarnos tiempo, yo mantengo que de verdad permiten escribir código con mayor rapidez, incluso en proyectos pequeños.

Por todas estas razones, todo el código del resto de este libro utilizará anotaciones de tipos. Supongo que algunos lectores se sentirán desanimados por utilizarlas, pero sospecho que al final del libro habrán cambiado de opinión.

Cómo escribir anotaciones de tipos

Como hemos visto, para tipos internos como `int`, `bool` y `float` basta con utilizar el propio tipo como anotación. Pero ¿qué pasa si tenemos (por

ejemplo) un `list`?

```
def total(xs: list) -> float:
   return sum(total)
```

Esto no es erróneo, pero el tipo no es lo bastante específico. Está claro que realmente queremos que `xs` sea un `list` de valores `float`, no (por ejemplo) un `list` de cadenas.

El módulo `typing` ofrece una serie de tipos parametrizados que podemos utilizar para hacer precisamente esto:

```
from typing import List      # observe la L mayúscula
def total(xs: List[float]) -> float:
   return sum(total)
```

Hasta ahora hemos especificado solamente anotaciones para parámetros de función y tipos de retorno. Para las propias variables suele ser obvio cuál es el tipo:

```
# Así es como se anota el tipo de variables cuando se definen.
# Pero esto es innecesario; es "obvio" que x es un int.
x: int = 5
```

No obstante, algunas veces no es obvio:

```
values = []             # ¿cuál es mi tipo?
best_so_far = None      # ¿cuál es mi tipo?
```

En estos casos suministraremos las comprobaciones de tipos *inline*:

```
from typing import Optional
values: List[int] = []
best_so_far: Optional[float] = None      # permitido ser un float o None
```

El módulo `typing` contiene muchos otros tipos, de los que solo emplearemos unos pocos:

```
# las anotaciones de tipo de este fragmento son todas innecesarias
from typing import Dict, Iterable, Tuple
```

```
# las claves son strings, los valores son ints
counts: Dict[str, int] = {‘data’: 1, ‘science’: 2}
# las listas y los generadores son ambos iterables
if lazy:
   evens: Iterable[int] = (x for x in range(10) if x % 2 == 0)
else:
   evens = [0, 2, 4, 6, 8]
# las tuplas especifican un tipo para cada elemento
triple: Tuple[int, float, int] = (10, 2.3, 5)
```

Finalmente, como Python tiene funciones de primera clase, necesitamos un tipo que las represente también. Este es un ejemplo bastante forzado:

```
from typing import Callable
# La comprobación de tipos dice que el repetidor es una función que admite
# dos argumentos, un string y un int, y devuelve un string.
def twice(repeater: Callable[[str, int], str], s: str) -> str:
   return repeater(s, 2)
def comma_repeater(s: str, n: int) -> str:
   n_copies = [s for _ in range(n)]
   return ‘, ‘.join(n_copies)
assert twice(comma_repeater, "type hints") == "type hints, type hints"
```

Como las anotaciones de tipos son solo objetos Python, podemos asignarles variables para que sea más fácil hacer referencia a ellos:

```
Number = int
Numbers = List[Number]
def total(xs: Numbers) -> Number:
   return sum(xs)
```

Para cuando lleguemos al final del libro, el lector estará bastante familiarizado con leer y escribir anotaciones de tipos, y espero que las utilice en su código.

Bienvenido a DataSciencester

Esto concluye la orientación al empleado. Ah, otra cosa: intente no

“distraer” nada.

Para saber más

- No faltan tutoriales de Python en el mundo. El oficial en https://docs.python.org/es/3/tutorial/ no es mal punto de partida para empezar.
- El tutorial oficial de IPython en http://ipython.readthedocs.io/en/stable/interactive/index.ht le permitirá empezar con IPython, si decide utilizarlo. Por favor, utilícelo.
- La documentación de `mypy` en https://mypy.readthedocs.io/en/stable/ le dará más información de la que nunca quiso tener sobre anotaciones de tipos y comprobaciones de tipos de Python.

---

[1] http://legacy.python.org/dev/peps/pep-0020/

[2] https://github.com/joelgrus/data-science-from-scratch/blob/master/INSTALL.md.

[3] https://www.python.org/.

[4] https://www.anaconda.com/products/distribution.

[5] https://docs.python.org/es/3/library/venv.html.

[6] https://virtualenv.pypa.io/en/latest/.

[7] http://ipython.org/.

3 Visualizar datos

Creo que la visualización es uno de los medios más poderosos de lograr objetivos personales.

—Harvey Mackay

La visualización de datos es una parte fundamental del kit de herramientas de un científico de datos. Es muy fácil crear visualizaciones, pero es mucho más difícil lograr que sean buenas. Tiene dos usos principales:

- Explorar datos.
- Comunicar datos.

En este capítulo, nos centraremos en adquirir las habilidades necesarias para empezar a explorar nuestros propios datos y producir las visualizaciones que vamos a utilizar a lo largo del libro. Al igual que la mayoría de los temas que se tratan en sus capítulos, la visualización de datos es un campo de estudio tan profundo que merece un libro entero. No obstante, trataré de darle una idea de lo que conduce a una buena visualización de datos y lo que no.

matplotlib

Existe una gran variedad de herramientas para visualizar datos. Emplearemos la librería de matplotlib[1], la más utilizada (aunque ya se le notan un poco los años). Si lo que queremos es producir una visualización elaborada e interactiva para la web, probablemente no es la mejor opción, pero sirve a la perfección para sencillos gráficos de barras, líneas y dispersión. Como ya mencioné anteriormente, matplotlib no es parte de la librería esencial de Python. Con el entorno virtual activado (para configurar uno, repase las instrucciones dadas en el apartado “Entornos virtuales” del capítulo 2), lo instalamos utilizando este comando:

```
python -m pip install matplotlib
```

Emplearemos el módulo `matplotlib.pyplot`. En su uso más sencillo, `pyplot` mantiene un estado interno en el que se crea una visualización paso a paso. En cuanto está lista, se puede guardar con `savefig` o mostrar con `show`.

Por ejemplo, hacer gráficos simples (como el de la figura 3.1) es bastante fácil:

```
from matplotlib import pyplot as plt
years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]
# crea un gráfico de líneas, años en el eje x, cantidades en el eje y
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')
# añade un título
plt.title("Nominal GDP")
# añade una etiqueta al eje y
plt.ylabel("Billions of $")
plt.show()
```

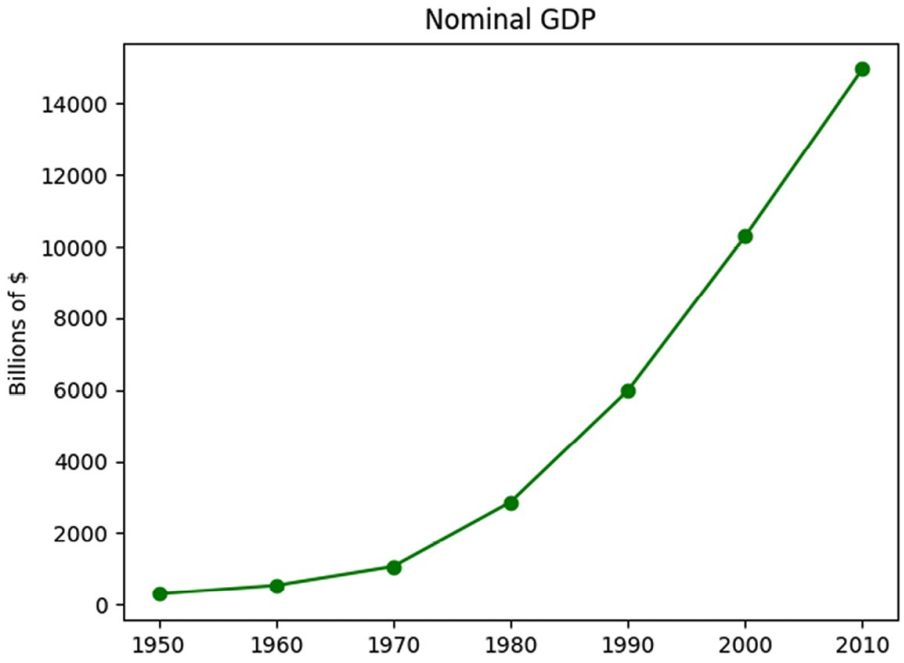


Figura 3.1. Un sencillo gráfico de líneas.

Crear gráficos con una calidad apta para publicaciones es más complicado,

y va más allá del objetivo de este capítulo. Hay muchas formas de personalizar los gráficos, por ejemplo, con etiquetas de ejes, estilos de línea y marcadores de puntos. En lugar de explicar estas opciones con todo detalle, simplemente utilizaremos algunas en nuestros ejemplos (y llamaré la atención sobre ello).

> **Nota:** Aunque no vayamos a utilizar mucho esta funcionalidad, matplotlib es capaz de producir complicados gráficos dentro de gráficos, aplicar formato de maneras sofisticadas y crear visualizaciones interactivas. En su documentación se puede encontrar información más detallada de la que ofrecemos en este libro.

Gráficos de barras

Un gráfico de barras es una buena elección cuando se desea mostrar cómo varía una cierta cantidad a lo largo de un conjunto discreto de elementos. Por ejemplo, la figura 3.2 muestra el número de Óscar que les fueron otorgados a cada una de una serie de películas:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]
# dibuja barras con coordenadas x de la izquierda [0, 1, 2, 3, 4], alturas
[num_oscars]
plt.bar(range(len(movies)), num_oscars)
plt.title("My Favorite Movies")                    # añade un título
plt.ylabel("# of Academy Awards")                  # etiqueta el eje y
# etiqueta el eje x con los nombres de las películas en el centro de las barras
plt.xticks(range(len(movies)), movies)
plt.show()
```

Un gráfico de barras también puede ser una buena opción para trazar histogramas de valores numéricos ordenados por cubos o *buckets*, como en la figura 3.3, con el fin de explorar visualmente el modo en que los valores están distribuidos:

```
from collections import Counter
```

```
grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]
# Agrupa las notas en bucket por decil, pero pone 100 con los 90
histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)
plt.bar([x + 5 for x in                    # Mueve barras a la derecha en 5
histogram.keys()],
     histogram.values(),                   # Da a cada barra su altura
                                           correcta
     10,                                   # Da a cada barra una anchura de 10
     edgecolor=(0, 0, 0))                  # Bordes negros para cada barra
plt.axis([-5, 105, 0, 5])                  # eje x desde -5 hasta 105,
                                           # eje y desde 0 hasta 5
plt.xticks([10 * i for i in                # etiquetas de eje x en 0, 10, ...,
range(11)])                                100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1
Grades")
plt.show()
```

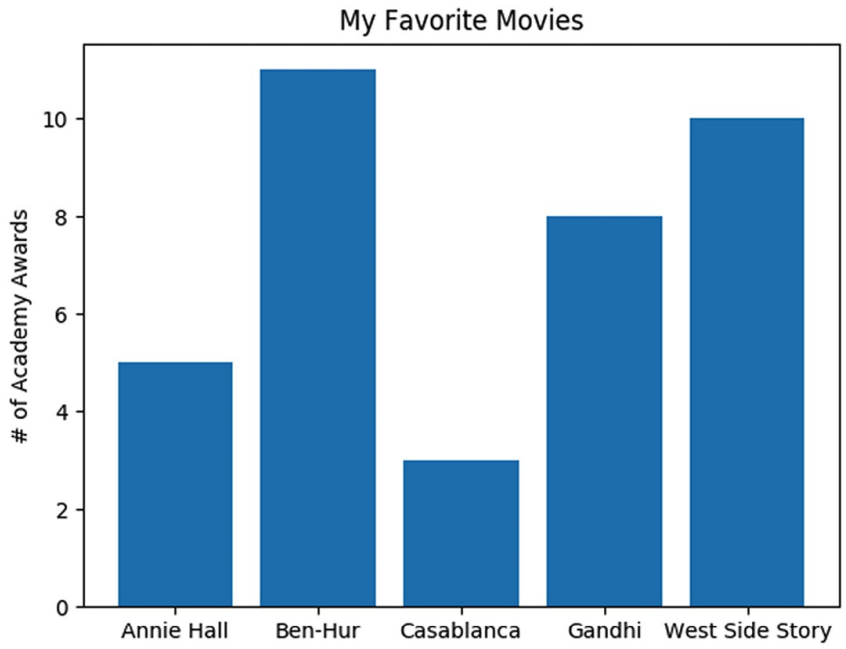


Figura 3.2. Un sencillo gráfico de barras.

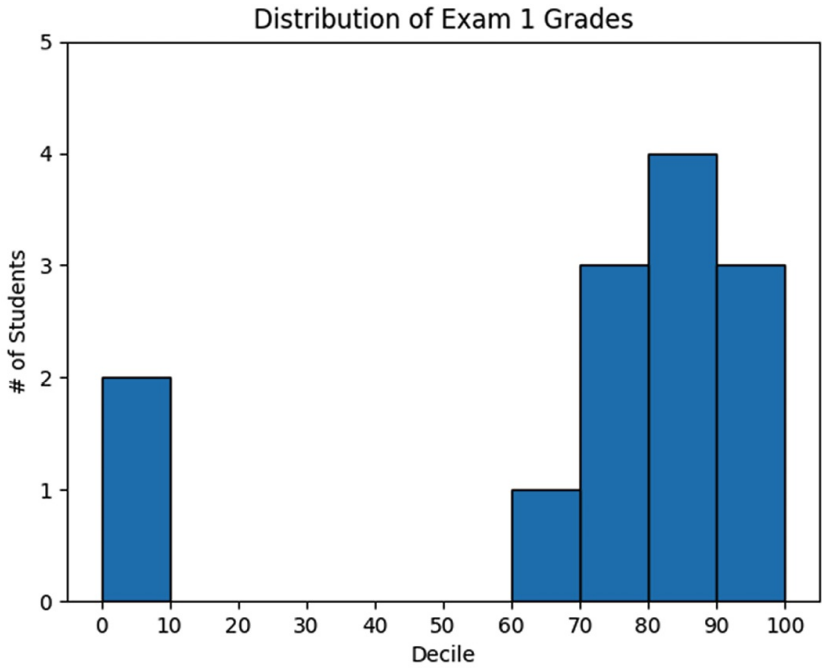


Figura 3.3. Utilizando un gráfico de barras para un histograma.

El tercer argumento de `plt.bar` especifica la anchura de las barras. Hemos elegido una anchura de 10, para llenar así todo el decil. También hemos desplazado las barras a la derecha en 5, de forma que, por ejemplo, la barra "10" (que corresponde al decil 10-20) tendría su centro en 15 y, por lo tanto, ocuparía el rango correcto. También añadimos un borde negro a cada barra para distinguirlas de forma visual.

La llamada a `plt.axis` indica que queremos que el eje x vaya desde -5 hasta 105 (solo para dejar un poco de espacio a la izquierda y a la derecha), y que el eje y varíe de 0 a 5; la llamada a `plt.xticks` pone las etiquetas del eje x en 0, 10, 20, ..., 100.

Conviene ser juiciosos al utilizar `plt.axis`. Cuando se crean gráficos de barras, está especialmente mal considerado que el eje y no empiece en 0, ya que de ese modo la gente se confunde con mucha facilidad (véase la figura 3.4):

```
mentions = [500, 505]
years = [2017, 2018]
plt.bar(years, mentions, 0.8)
```

```
plt.xticks(years)
plt.ylabel("# of times I heard someone say ‘data science’")
# si no se hace esto, matplotlib etiquetará el eje x con 0, 1
# y añadirá +2.013e3 en la esquina (¡qué malo es matplotlib!)
plt.ticklabel_format(useOffset=False)
# el eje y erróneo solo muestra la parte sobre 500
plt.axis([2016.5, 2018.5, 499, 506])
plt.title("Look at the ‘Huge’ Increase!")
plt.show()
```

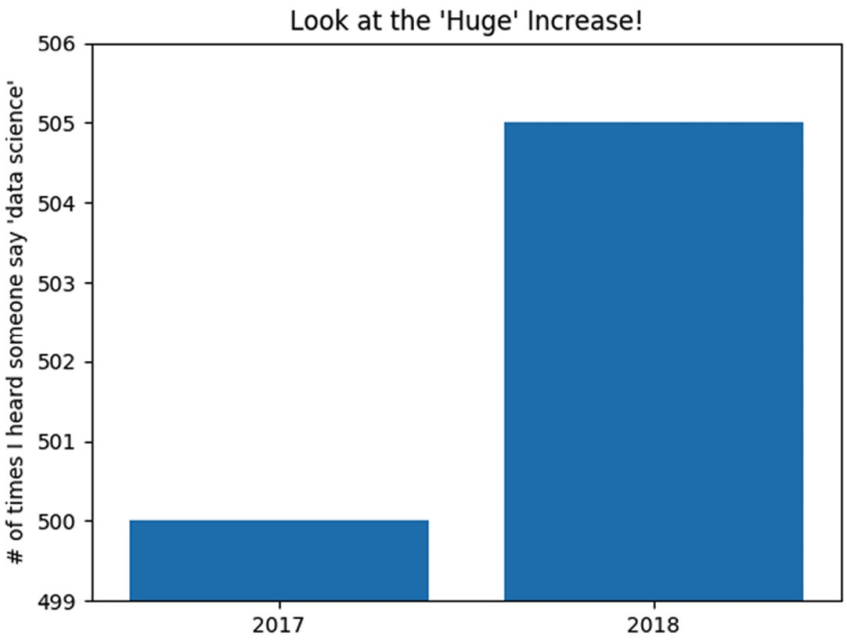


Figura 3.4. Un gráfico con el eje y erróneo.

En la figura 3.5 utilizamos ejes más sensatos, aunque así no queda tan impresionante:

```
plt.axis([2016.5, 2018.5, 0, 550])
plt.title("Not So Huge Anymore")
plt.show()
```

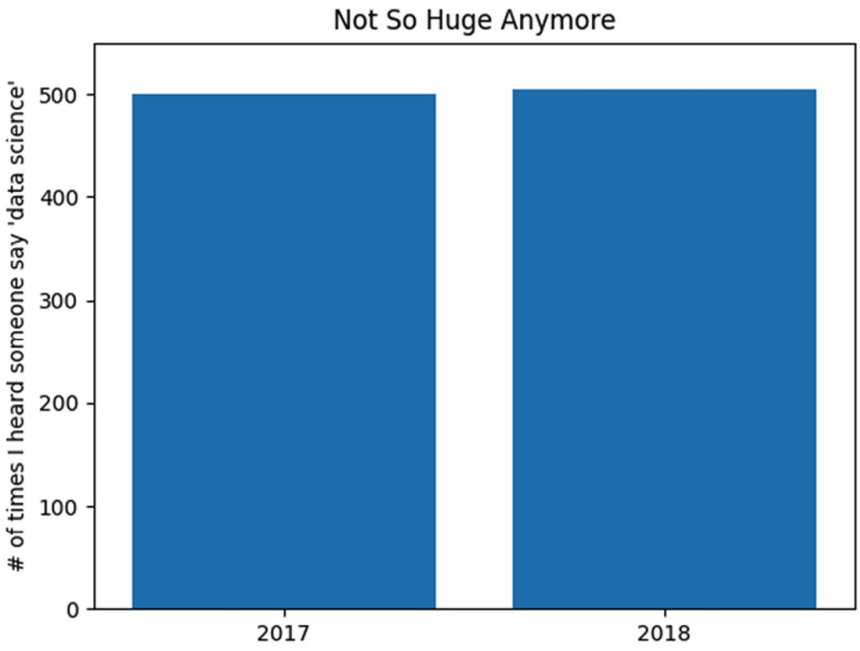


Figura 3.5. El mismo gráfico con un eje y nada confuso.

Gráficos de líneas

Como ya hemos visto, podemos hacer gráficos de líneas utilizando `plt.plot`. Son una buena elección para mostrar tendencias, como se ilustra en la figura 3.6:

```
variance = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]
# Podemos hacer varias llamadas a plt.plot
# para mostrar varias series en el mismo gráfico
plt.plot(xs, variance, ‘g-’, label=’variance’)      # línea continua
plt.plot(xs, bias_squared, ‘r-.’, label=’bias^2’)   # línea de puntos y guiones
plt.plot(xs, total_error, ‘b:’, label=’total error’)# línea de puntos
# Como asignamos etiquetas a cada serie,
# obtenemos una leyenda gratis (loc=9 significa "arriba centro")
plt.legend(loc=9)
```

```
plt.xlabel("model complexity")
plt.xticks([])
plt.title("The Bias-Variance Tradeoff")
plt.show()
```

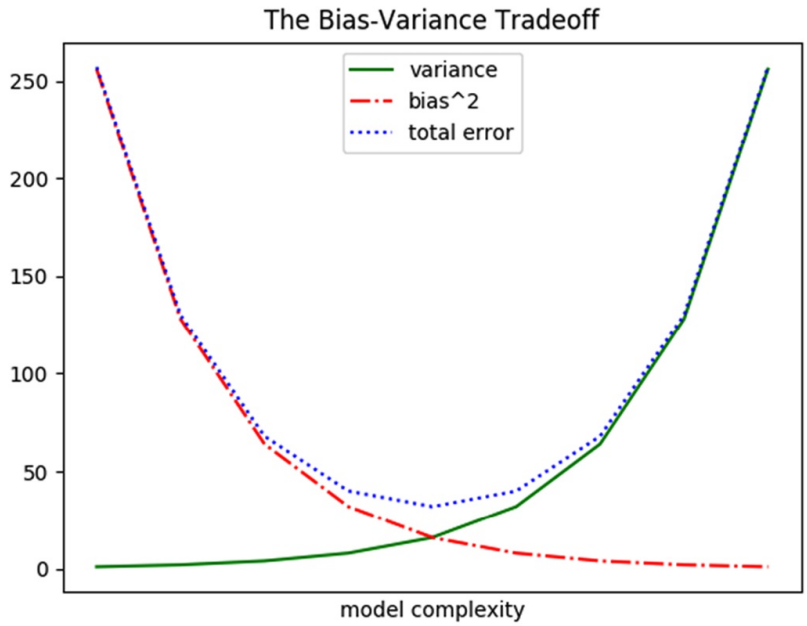


Figura 3.6. Varios gráficos de líneas con una leyenda.

Gráficos de dispersión

Un gráfico de dispersión es la opción adecuada para visualizar la relación entre dos conjuntos de datos emparejados. Por ejemplo, la figura 3.7 ilustra la relación entre el número de amigos que tienen sus usuarios y el número de minutos que pasan cada día en el sitio:

```
friends = [ 70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
plt.scatter(friends, minutes)
# etiqueta cada punto
for label, friend_count, minute_count in zip(labels, friends, minutes):
   plt.annotate(label,
```

```
    xy=(friend_count, minute_count),      # Pone la etiqueta con su punto
    xytext=(5, -5),                       # pero un poco desplazada
    textcoords='offset points')
plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```

Si estamos representando variables comparables, podríamos obtener una imagen confusa si dejáramos que matplotlib eligiera la escala, como en la figura 3.8.

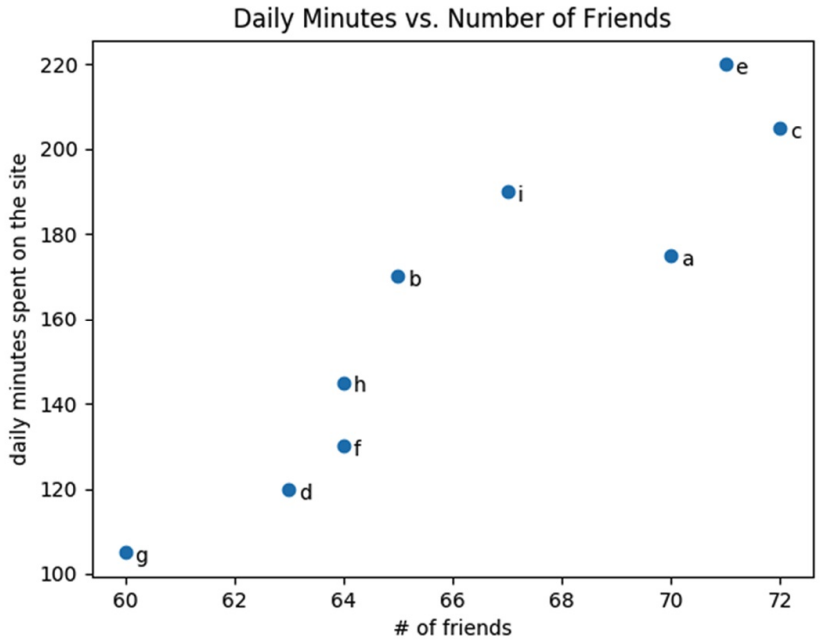


Figura 3.7. Un gráfico de dispersión de amigos y tiempo en el sitio.

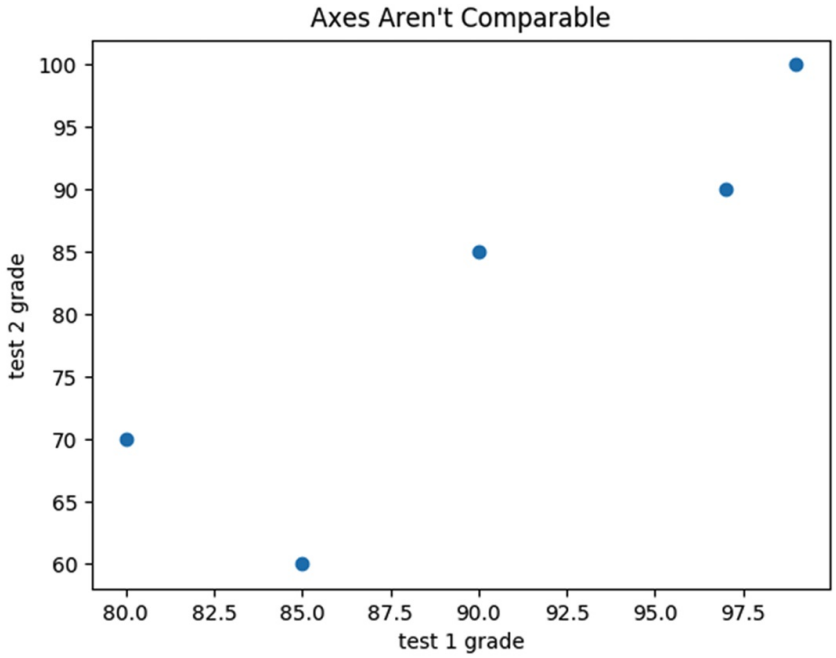


Figura 3.8. Un gráfico de dispersión con ejes imposibles de comparar.

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]
plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren’t Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```

Si incluimos una llamada a `plt.axis(“equal”)`, el gráfico (figura 3.9) muestra con mayor precisión que la mayor parte de la variación tiene lugar en la prueba 2.

Con esto basta para empezar con la visualización. Aprenderemos mucho más sobre la visualización a lo largo del libro.

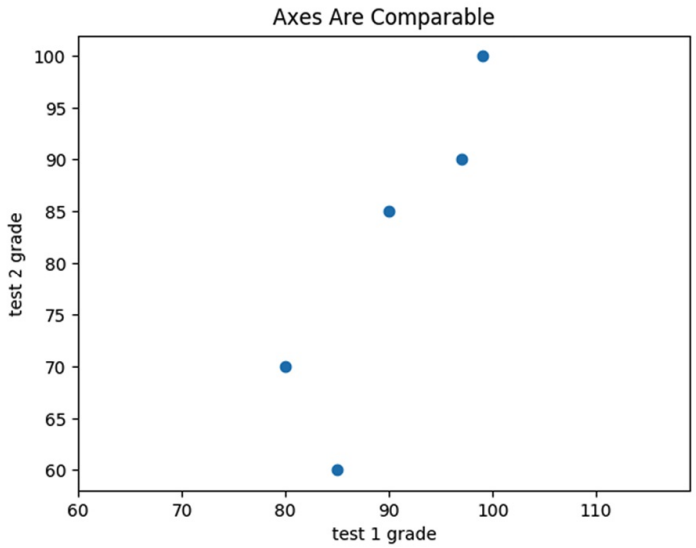


Figura 3.9. El mismo gráfico de dispersión con ejes iguales.

Para saber más

- La galería de matplotlib, en `https://matplotlib.org/stable/gallery/index.html`, da una idea bastante buena del tipo de cosas que se pueden hacer con matplotlib (y de cómo hacerlas).
- seaborn, en `https://seaborn.pydata.org/`, se ha creado en base a matplotlib y permite producir fácilmente visualizaciones más bonitas (y complejas).
- Altair, en `https://altair-viz.github.io/`, es una nueva librería de Python para crear visualizaciones declarativas.
- D3.js, en `https://d3js.org`, es una librería de JavaScript para producir sofisticadas visualizaciones interactivas para la web. Aunque no está en Python, se utiliza mucho y vale la pena familiarizarse con ella.
- Bokeh, en `https://bokeh.pydata.org`, es una librería que permite incorporar a Python visualizaciones de estilo D3.

[1] https://matplotlib.org/.

4 Álgebra lineal

¿Hay algo más inútil o menos útil que el álgebra?

—Billy Conolly

El álgebra lineal es la rama de las matemáticas que se ocupa de los espacios vectoriales. Aunque no espero que el lector aprenda álgebra lineal en un breve capítulo, se apoya en un gran número de conceptos y técnicas de ciencia de datos, lo que significa que al menos les debo un intento. Lo que vamos a aprender en este capítulo lo utilizaremos mucho a lo largo del libro.

Vectores

Definidos de una forma abstracta, los vectores son objetos que se pueden sumar para formar nuevos vectores y se pueden multiplicar por escalares (es decir, números), también para formar nuevos vectores.

De una forma más concreta (para nosotros), digamos que los vectores son puntos de un espacio de dimensión finita. Aunque no se nos suele ocurrir pensar en los datos como vectores, a menudo son una forma útil de representar datos numéricos.

Por ejemplo, si tenemos las alturas, pesos y edades de un gran número de personas, podemos tratar los datos como vectores tridimensionales `[height, weight, age]`. Si tuviéramos una clase con cuatro exámenes, podríamos tratar las notas de los alumnos como vectores de cuatro dimensiones `[exam1, exam2, exam3, exam4]`.

El enfoque más sencillo para aprender esto desde cero es representar los vectores como una lista de números. Una lista de tres números corresponde a un vector en un espacio tridimensional y viceversa.

Realizaremos esto con un alias de tipo que dice que un `Vector` es solo una `list` de valores de tipo `float`:

```
from typing import List
Vector = List[float]
height_weight_age = [70,         # pulgadas,
                     170,        # libras,
                     40 ]        # años
grades = [95,      # examen1
80,                # examen2
75,                # examen3
62 ]               # examen4
```

También querremos realizar aritmética con los vectores. Como las `list` de Python no son vectores (y por lo tanto no dan facilidades para la aritmética con vectores), tendremos que crear nosotros mismos estas herramientas de aritmética. Así que vamos allá.

Para empezar, muy a menudo necesitaremos sumar dos vectores. Los vectores se suman componente a componente, lo que significa que, si dos vectores `v` y `w` tienen la misma longitud, su suma es sencillamente el vector cuyo primer elemento es `v[0] + w[0]`, cuyo segundo elemento es `v[1] + w[1]`, y así sucesivamente (si no tienen la misma longitud, entonces no se pueden sumar). Por ejemplo, sumar los vectores `[1, 2]` y `[2, 1]` da como resultado `[1 + 2, 2 + 1]` o `[3,3]`, como muestra la figura 4.1.

Podemos implementar esto fácilmente comprimiendo los vectores con `zip` y utilizando una comprensión de lista para sumar los elementos correspondientes:

```
def add(v: Vector, w: Vector) -> Vector:
   """Adds corresponding elements"""
   assert len(v) == len(w), "vectors must be the same length"
   return [v_i + w_i for v_i, w_i in zip(v, w)]
assert add([1, 2, 3], [4, 5, 6]) == [5, 7, 9]
```

De forma similar, para restar dos vectores simplemente restamos los elementos correspondientes:

```
def subtract(v: Vector, w: Vector) -> Vector:
   """Subtracts corresponding elements"""
   assert len(v) == len(w), "vectors must be the same length"
   return [v_i–w_i for v_i, w_i in zip(v, w)]
```

```
assert subtract([5, 7, 9], [4, 5, 6]) == [1, 2, 3]
```

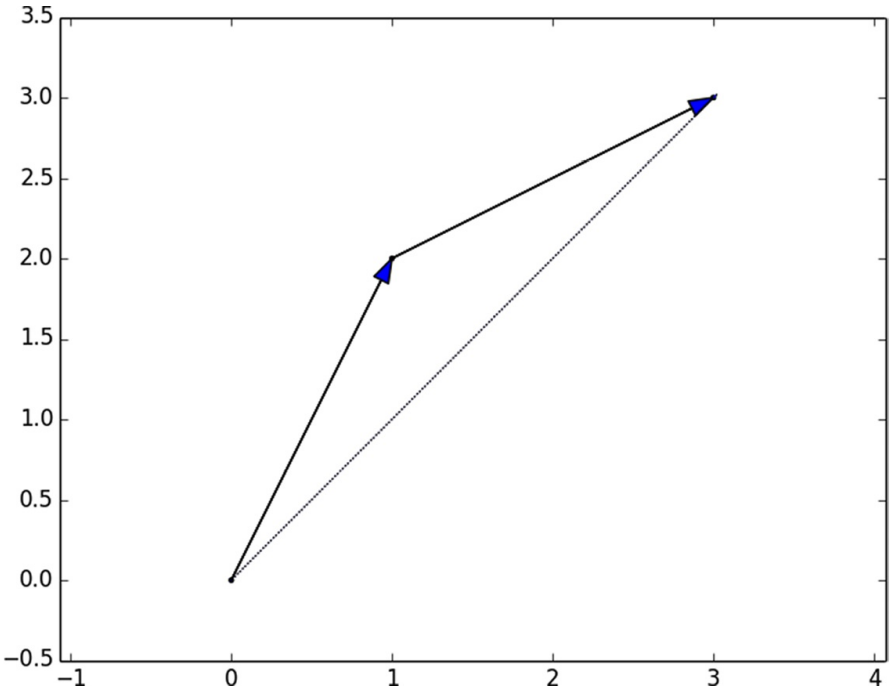


Figura 4.1. Sumando dos vectores.

También querremos en ocasiones sumar una lista de vectores por componentes (es decir, crear un nuevo vector cuyo primer elemento es la suma de todos los primeros elementos y cuyo segundo elemento es la suma de todos los segundos elementos, y así sucesivamente):

```
def vector_sum(vectors: List[Vector]) -> Vector:
   """Sums all corresponding elements"""
   # Comprueba que los vectores no estén vacíos
   assert vectors, "no vectors provided!"
   # Comprueba que los vectores tienen el mismo tamaño
   num_elements = len(vectors[0])
   assert all(len(v) == num_elements for v in vectors), "different sizes!"
   # el elemento i del resultado es la suma de cada vector [i]
   return [sum(vector[i] for vector in vectors)
        for i in range(num_elements)]
assert vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]) == [16, 20]
```

También tendremos que ser capaces de multiplicar un vector por un escalar, cosa que hacemos sencillamente multiplicando cada elemento del

vector por dicho número:

```
def scalar_multiply(c: float, v: Vector) -> Vector:
     """Multiplies every element by c"""
     return [c * v_i for v_i in v]
assert scalar_multiply(2, [1, 2, 3]) == [2, 4, 6]
```

Esto nos permite calcular la media por componentes de una lista de vectores (del mismo tamaño):

```
def vector_mean(vectors: List[Vector]) -> Vector:
   """Computes the element-wise average"""
   n = len(vectors)
   return scalar_multiply(1/n, vector_sum(vectors))
assert vector_mean([[1, 2], [3, 4], [5, 6]]) == [3, 4]
```

Una herramienta menos obvia es el producto punto. El producto punto de dos vectores es la suma de los productos de sus componentes:

```
def dot(v: Vector, w: Vector) -> float:
   """Computes v_1 * w_1 + ... + v_n * w_n"""
   assert len(v) == len(w), "vectors must be same length"
   return sum(v_i * w_i for v_i, w_i in zip(v, w))
assert dot([1, 2, 3], [4, 5, 6]) == 32     # 1 * 4 + 2 * 5 + 3 * 6
```

Si `w` tiene magnitud 1, el producto punto mide hasta dónde se extiende el vector `v` en la dirección `w`. Por ejemplo, si `w = [1, 0]`, entonces `dot(v, w)` es el primer componente de `v`. Otra forma de decir esto es que es la longitud del vector que se obtendría si se proyectara `v` en `w` (véase la figura 4.2).

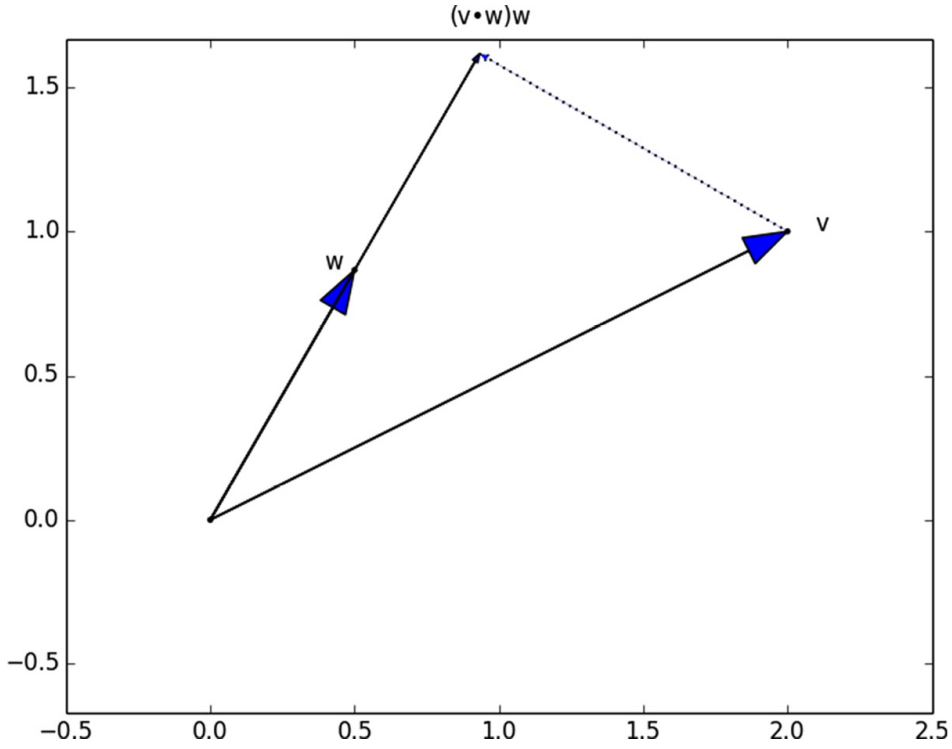


Figura 4.2. El producto punto como proyección de vector.

Utilizando esto, es fácil calcular la suma de cuadrados de un vector:

```
def sum_of_squares(v: Vector) -> float:
   """Returns v_1 * v_1 + ... + v_n * v_n"""
   return dot(v, v)
assert sum_of_squares([1, 2, 3]) == 14 # 1 * 1 + 2 * 2 + 3 * 3
```

Que podemos utilizar para calcular su magnitud (o longitud):

```
import math
def magnitude(v: Vector) -> float:
     """Returns the magnitude (or length) of v"""
     return                                  # math.sqrt es la función raíz
     math.sqrt(sum_of_squares(v))            cuadrada
assert magnitude([3, 4]) == 5
```

Ahora tenemos todas las piezas necesarias para calcular la distancia entre dos vectores, definida como:

$$\sqrt{(v_1 - w_1)^2 + ... + (v_n - w_n)^2}$$

En código:

```
def squared_distance(v: Vector, w: Vector) -> float:
   """Computes (v_1-w_1) ** 2 + ... + (v_n-w_n) ** 2"""
   return sum_of_squares(subtract(v, w))
def distance(v: Vector, w: Vector) -> float:
   """Computes the distance between v and w"""
   return math.sqrt(squared_distance(v, w))
```

Quizá esto quede más claro si lo escribimos como (el equivalente):

```
def distance(v: Vector, w: Vector) -> float:
   return magnitude(subtract(v, w))
```

Esto debería ser suficiente para empezar. Vamos a utilizar muchísimo estas funciones a lo largo del libro.

> **Nota:** Utilizar listas como vectores es excelente de cara a la galería, pero terrible para el rendimiento.
> En código de producción, nos interesará más utilizar la librería NumPy, que incluye una clase de objetos *array* de alto rendimiento con todo tipo de operaciones aritméticas incluidas.

Matrices

Una matriz es una colección de números bidimensional. Representaremos las matrices como listas de listas, teniendo cada lista interior el mismo tamaño y representando una fila de la matriz. Si `A` es una matriz, entonces `A[i][j]` es el elemento de la fila *i* y la columna *j*. Por convenio matemático, utilizaremos con frecuencia letras mayúsculas para representar matrices. Por ejemplo:

```
# Otro alias de tipo
Matrix = List[List[float]]
A = [[1, 2, 3],      # A tiene 2 filas y 3 columnas
   [4, 5, 6]]
B = [[1, 2],         # B tiene 3 filas y 2 columnas
   [3, 4],
   [5, 6]]
```

Nota: En matemáticas, se suele denominar a la primera fila de la matriz “fila 1” y a la primera columna “columna 1”. Como estamos representando matrices con `list` de Python, que están indexadas al 0, llamaremos a la primera fila de la matriz “fila 0” y a la primera columna “columna 0”.

Dada esta representación de lista de listas, la matriz `A` tiene `len(A)` filas y `len(A[0])` columnas, lo que consideramos su `shape`:

```
from typing import Tuple
def shape(A: Matrix) -> Tuple[int, int]:
   """Returns (# of rows of A,               # of columns of A)"""
   num_rows = len(A)
   num_cols = len(A[0]) if A else 0          # número de elementos de la
                                             primera fila
   return num_rows, num_cols
assert shape([[1, 2, 3], [4, 5, 6]]) ==      # 2 filas, 3 columnas
(2, 3)
```

Si una matriz tiene *n* filas y *k* columnas, nos referiremos a ella como una matriz *n* x *k*. Podemos (y a veces lo haremos) pensar en cada fila de una matriz *n* x *k* como un vector de longitud *k*, y en cada columna como en un vector de longitud *n*:

```
def get_row(A: Matrix, i: int) -> Vector:
   """Returns the i-th row of A (as a Vector)"""
   return A[i]            # A[i] es ya la fila i
def get_column(A: Matrix, j: int) -> Vector:
   """Returns the j-th column of A (as a Vector)"""
   return [A_i[j]         # elemento j de la fila A_i
        for A_i in A]     # para cada fila A_i
```

También querremos poder crear una matriz dada su forma y una función

para generar sus elementos. Podemos hacer esto utilizando una comprensión de lista anidada:

```
from typing import Callable
def make_matrix(num_rows: int,
                num_cols: int,
                entry_fn: Callable[[int, int], float]) -> Matrix:
   """
   Returns a num_rows x num_cols matrix
   whose (i,j)-th entry is entry_fn(i, j)
   """
   return [[entry_fn(i, j)             # dado i, crea una lista
        for j in range(num_cols)]       # [entry_fn(i, 0), ... ]
        for i in range(num_rows)]      # crea una lista por cada i
```

Dada esta función, se podría crear una matriz identidad 5 x 5 (con unos en la diagonal y ceros en el resto) de esta forma:

```
def identity_matrix(n: int) -> Matrix:
   """Returns the n x n identity matrix"""
   return make_matrix(n, n, lambda i, j: 1 if i == j else 0)
assert identity_matrix(5) ==          [[1, 0, 0, 0, 0],
                                       [0, 1, 0, 0, 0],
                                       [0, 0, 1, 0, 0],
                                       [0, 0, 0, 1, 0],
                                       [0, 0, 0, 0, 1]]
```

Las matrices serán importantes para nosotros por varias razones.

En primer lugar, podemos utilizar una matriz para representar un conjunto de datos formado por varios vectores, simplemente considerando cada vector como una fila de la matriz. Por ejemplo, si tuviéramos las alturas, pesos y edades de 1.000 personas, podríamos poner estos datos en una matriz 1.000 x 3:

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
      ]
```

En segundo lugar, como veremos más tarde, podemos utilizar una matriz *n*

x *k* para representar una función lineal que transforme vectores de *k* dimensiones en vectores de *n* dimensiones. Varias de nuestras técnicas y conceptos implicarán tales funciones.

Tercero, las matrices se pueden utilizar para representar relaciones binarias. En el capítulo 1, representamos los bordes de una red como una colección de pares `(i, j)`. Una representación alternativa sería crear una matriz `A` tal que `A[i][j]` sea 1 si los nodos `i` y `j` están conectados y 0 en otro caso.

Recordemos que antes teníamos:

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

También podemos representar esto como:

```
#                                          usuario 0 1 2 3 4 5 6 7 8 9
#
friend_matrix = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0],          # usuario 0
                 [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],          # usuario 1
                 [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],          # usuario 2
                 [0, 1, 1, 0, 1, 0, 0, 0, 0, 0],          # usuario 3
                 [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],          # usuario 4
                 [0, 0, 0, 0, 1, 0, 1, 1, 0, 0],          # usuario 5
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],          # usuario 6
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],          # usuario 7
                 [0, 0, 0, 0, 0, 0, 1, 1, 0, 1],          # usuario 8
                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]]          # usuario 9
```

Si hay pocas conexiones, esta representación es mucho menos eficaz, ya que terminamos teniendo que almacenar muchos ceros. No obstante, con la representación en matriz se comprueba mucho más rápido si dos nodos están conectados (basta con hacer una búsqueda en la matriz en lugar de, posiblemente, inspeccionar cada borde):

```
assert friend_matrix[0][2] == 1, "0 and 2 are friends"
assert friend_matrix[0][8] == 0, "0 and 8 are not friends"
```

De forma similar, para encontrar las conexiones de un nodo, basta con inspeccionar la columna (o la fila) correspondiente a ese nodo:

```
# basta con mirar en una fila
friends_of_five = [i
             for i, is_friend in enumerate(friend_matrix[5])
             if is_friend]
```

Con un gráfico de pequeño tamaño se podría añadir una lista de conexiones a cada objeto de nodo para acelerar este proceso, pero, con uno más grande y en constante evolución, hacer esto sería probablemente demasiado caro y difícil de mantener.

Revisaremos las matrices a lo largo del libro.

Para saber más

- Los científicos de datos utilizan mucho el álgebra lineal (lo que se da por sentado con frecuencia, aunque no con tanta por personas que no lo comprenden). No sería mala idea leer un libro de texto. Se pueden encontrar varios disponibles gratuitamente en línea:
  - *Linear Algebra*, en http://joshua.smcvt.edu/linearalgebra/, de Jim Hefferon (Saint Michael's College).
  - *Linear Algebra*, en https://www.math.ucdavis.edu/~linear/linear-guest.pdf, de David Cherney, Tom Denton, Rohit Thomas y Andrew Waldron (UC Davis).
  - Si se siente aventurero, *Linear Algebra Done Wrong*, en https://www.math.brown.edu/~treil/papers/LADW/LADW_2017-09-04.pdf, de Sergei Treil (Brown University), es una introducción más avanzada.
- Toda la maquinaria que hemos creado en este capítulo se puede conseguir de forma gratuita utilizando NumPy, en http://www.numpy.org (también se obtiene mucho más, incluyendo mucho mejor rendimiento).

5 Estadística

Los hechos son obstinados, pero las estadísticas son más maleables.

—Mark Twain

El término estadística hace referencia a las matemáticas y a las técnicas con las que comprendemos los datos. Es un campo rico y extenso, más adecuado para una estantería (o una sala entera) de una biblioteca que para un capítulo de un libro, de modo que necesariamente mi exposición no podrá ser profunda. Trataré de enseñar lo justo para que resulte comprometido y active el interés lo suficiente como para seguir adelante y aprender aún más.

Describir un solo conjunto de datos

Mediante una combinación de boca a boca y suerte, DataSciencester ha crecido y ahora tiene muchísimos miembros, y el vicepresidente de Recaudación de fondos quiere algún tipo de descripción de la cantidad de amigos que tienen sus miembros para poder incluirla en sus discursos de presentación.

Utilizando las técnicas del capítulo 1, es muy sencillo producir estos datos. Pero ahora nos enfrentamos al problema de cómo describirlos. Una descripción obvia de cualquier conjunto de datos es sencillamente los propios datos:

```
num_friends = [100, 49, 41, 40, 25,
          # ... y muchos más
          ]
```

Con un conjunto de datos bastante pequeño, esta podría ser la mejor descripción. Pero, con otro más grande, resulta difícil de manejar y probablemente opaco (imagínese tener que mirar fijamente una lista de 1

millón de números). Por esta razón, utilizamos las estadísticas para sintetizar y comunicar las características relevantes de nuestros datos. Como primer enfoque, ponemos los contadores de amigos en un histograma utilizando `Counter` y `plt.bar` (véase la figura 5.1):

```
from collections import Counter
import matplotlib.pyplot as plt
friend_counts = Counter(num_friends)
xs = range(101)                         # el valor mayor es 100
ys = [friend_counts[x] for x in         # la altura es justamente núm. de
xs]                                     amigos
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
plt.show()
```

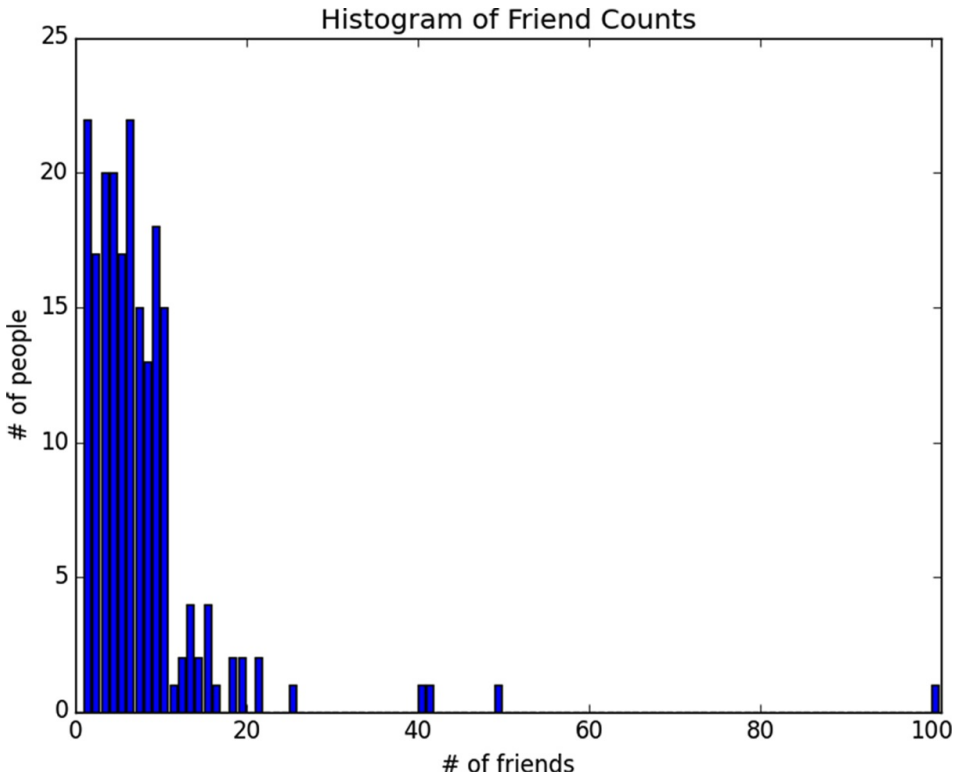


Figura 5.1. Un histograma de contadores de amigos.

Por desgracia, sigue siendo demasiado difícil meter este gráfico en las conversaciones. De modo que empezamos a generar algunas estadísticas;

probablemente la más sencilla es el número de puntos de datos:

```
num_points = len(num_friends)      # 204
```

Probablemente también estemos interesados en los valores mayor y menorww:

```
largest_value = max(num_friends)       # 100
smallest_value = min(num_friends)      # 1
```

Que son simplemente casos especiales de querer saber cuáles son los valores de determinadas posiciones:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]             # 1
second_smallest_value = sorted_values[1]      # 1
second_largest_value = sorted_values[-2]      # 49
```

Pero solo estamos empezando.

Tendencias centrales

Normalmente, querremos tener alguna noción del lugar en el que nuestros datos están centrados. Lo más habitual es que usemos la media (o promedio), que no es más que la suma de los datos dividida por el número de datos:

```
def mean(xs: List[float]) -> float:
   return sum(xs) / len(xs)
mean(num_friends)       # 7,333333
```

Si tenemos dos puntos de datos, la media es simplemente el punto a mitad de camino entre los dos. A medida que se añaden más puntos, la media se va desplazando, pero siempre depende del valor de cada punto. Por ejemplo, si tenemos 10 puntos de datos y aumentamos el valor de cualquiera de ellos en 1, la media aumenta en 0,1.

También estaremos interesados en ocasiones en la mediana, que es el valor más céntrico (si el número de puntos de datos es impar) o el promedio

de los dos valores más céntricos (si el número de puntos de datos es par).

Por ejemplo, si tenemos cinco puntos de datos en un vector ordenado `x`, la mediana es `x[5 // 2]` o `x[2]`. Si tenemos seis puntos de datos, queremos el promedio de `x[2]` (el tercer punto) y `x[3]` (el cuarto punto).

Tengamos en cuenta que (a diferencia de la media) la mediana no depende por completo de cada valor de los datos. Por ejemplo, si el punto más grande lo hacemos aún mayor (o el punto más pequeño menor), los puntos medios no cambian, lo que significa que la mediana sí lo hace.

Escribiremos distintas funciones para los casos par e impar y las combinaremos:

```
# Los guiones bajos indican que son funciones "privadas", destinadas a
# ser llamadas por nuestra función mediana pero no por otras personas
# que utilicen nuestra librería de estadísticas.
def _median_odd(xs: List[float]) -> float:
   """If len(xs) is odd, the median is the middle element"""
   return sorted(xs)[len(xs) // 2]
def _median_even(xs: List[float]) -> float:
   """If len(xs) is even, it's the average of the middle two elements"""
   sorted_xs = sorted(xs)
   hi_midpoint = len(xs) // 2      # p.ej. longitud 4 => hi_midpoint 2
   return (sorted_xs[hi_midpoint–1] + sorted_xs[hi_midpoint]) / 2
def median(v: List[float]) -> float:
   """Finds the 'middle-most' value of v"""
   return _median_even(v) if len(v) % 2 == 0 else _median_odd(v)
assert median([1, 10, 2, 9, 5]) == 5
assert median([1, 9, 2, 10]) == (2 + 9) / 2
```

Y ahora podemos calcular el número medio de amigos:

```
print(median(num_friends))     # 6
```

Sin duda, la media es más sencilla de calcular y varía ligeramente cuando nuestros datos cambian. Si tenemos *n* puntos de datos y uno de ellos aumenta en una cierta pequeña cantidad *e*, entonces la media necesariamente aumentará en *e* / *n* (lo que consigue que la media sea susceptible a todo tipo de trucos de cálculo). Pero, para hallar la mediana, tenemos que ordenar los datos. Y cambiar uno de nuestros puntos de datos en una pequeña cantidad *e*

podría aumentar la mediana también en *e*, en una cantidad inferior a *e* o en nada en absoluto (dependiendo del resto de datos).

> **Nota:** En realidad, existen trucos no evidentes para calcular medianas eficazmente[1] sin ordenar los datos. Sin embargo, están más allá del objetivo de este libro, de modo que toca ordenar los datos.

Al mismo tiempo, la media es muy sensible a los valores atípicos de nuestros datos. Si nuestro usuario más sociable tuviera 200 amigos (en lugar de 100), entonces la media subiría a 7,82, mientras que la mediana seguiría siendo la misma. Si es probable que los valores atípicos sean datos erróneos (o no representativos del fenómeno que estemos tratando de comprender), entonces la media puede darnos a veces una imagen equívoca. Por ejemplo, a menudo se cuenta la historia de que, a mediados de los años 80, la asignatura de la Universidad de Carolina del Norte con el salario inicial medio más alto era geografía, principalmente debido a la estrella de la NBA (y valor atípico) Michael Jordan.

Una generalización de la mediana es el cuantil, que representa el valor bajo el cual reside un determinado percentil de los datos (la mediana representa el valor bajo el cual reside el 50 % de los datos):

```
def quantile(xs: List[float], p: float) -> float:
   """Returns the pth-percentile value in x"""
   p_index = int(p * len(xs))
   return sorted(xs)[p_index]
assert quantile(num_friends, 0.10) == 1
assert quantile(num_friends, 0.25) == 3
assert quantile(num_friends, 0.75) == 9
assert quantile(num_friends, 0.90) == 13
```

Menos habitual sería que quisiéramos mirar la moda, es decir, el valor o valores más comunes:

```
def mode(x: List[float]) -> List[float]:
   """Returns a list, since there might be more than one mode"""
   counts = Counter(x)
   max_count = max(counts.values())
```

```
    return [x_i for x_i, count in counts.items()
            if count == max_count]
assert set(mode(num_friends)) == {1, 6}
```

Pero lo más habitual es que utilicemos la media.

Dispersión

Dispersión se refiere a las medidas de dispersión de nuestros datos. Normalmente son estadísticas para las que los valores cercanos a cero significan “no disperso en absoluto” y para las que los valores grandes (lo que sea que eso signifique) quieren decir “muy disperso”. Por ejemplo, una medida muy sencilla es el rango, que no es otra cosa que la diferencia entre los elementos mayor y menor:

```
# "range" ya significa algo en Python, así que usaremos otro nombre
def data_range(xs: List[float]) -> float:
    return max(xs)-min(xs)
assert data_range(num_friends) == 99
```

El rango es cero precisamente cuando el `max` y el `min` son iguales, cosa que solo puede ocurrir si los elementos de `x` son todos iguales, lo que significa que los datos están tan poco dispersos como es posible. A la inversa, si el rango es grande, entonces el `max` es mucho más grande que el `min` y los datos están más dispersos.

Al igual que ocurre con la mediana, en realidad el rango no depende del conjunto de datos entero. Un conjunto de datos cuyos puntos son todos 0 o 100 tiene el mismo rango que otro cuyos valores sean 0, 100 y muchos 50. Pero parece que el primer conjunto de datos “debería” estar más disperso.

Una medida más compleja de dispersión es la varianza, que se calcula como:

```
from scratch.linear_algebra import sum_of_squares
def de_mean(xs: List[float]) -> List[float]:
    """Translate xs by subtracting its mean (so the result has mean 0)"""
    x_bar = mean(xs)
    return [x-x_bar for x in xs]
```

```
def variance(xs: List[float]) -> float:
   """Almost the average squared deviation from the mean"""
   assert len(xs) >= 2, "variance requires at least two elements"
   n = len(xs)
   deviations = de_mean(xs)
   return sum_of_squares(deviations) / (n–1)
assert 81.54 < variance(num_friends) < 81.55
```

Nota: Parece que esto sea casi la desviación cuadrática promedio respecto a la media, salvo que estamos dividiendo por `n - 1` en lugar de por `n`. De hecho, cuando tratamos con una muestra de una población mayor, `x_bar` es solamente una estimación de la media real, lo que significa que en promedio `(x_i - x_bar) ** 2` es una subestimación de la desviación cuadrática de `x_i` con respecto a la media, razón por la cual dividimos por `n - 1` en lugar de por `n`. Consulte la Wikipedia.[2]

Ahora, sin importar en qué unidades estén nuestros datos (por ejemplo, “amigos”), todas nuestras medidas de tendencia central están en la misma unidad, igual que el rango. Pero la varianza, por otro lado, tiene unidades que son el cuadrado de las originales (es decir, “amigos al cuadrado”). Como puede resultar difícil darle sentido a esto, a menudo recurrimos en su lugar a la desviación estándar:

```
import math
def standard_deviation(xs: List[float]) -> float:
   """The standard deviation is the square root of the variance"""
   return math.sqrt(variance(xs))
assert 9.02 < standard_deviation(num_friends) < 9.04
```

Tanto el rango como la desviación estándar tienen el mismo problema de valor atípico que vimos antes con la media. Utilizando el mismo ejemplo, si nuestro usuario más sociable tuviera realmente 200 amigos, la desviación estándar sería 14,89 (¡más del 60 % más elevada!).

Una alternativa más robusta calcula la diferencia entre los percentiles 75 y 25:

```
def interquartile_range(xs: List[float]) -> float:
   """Returns the difference between the 75%-ile and the 25%-ile"""
   return quantile(xs, 0.75)–quantile(xs, 0.25)
```

```
assert interquartile_range(num_friends) == 6
```

Que apenas se ve afectada por una pequeña cantidad de valores atípicos.

Correlación

La vicepresidenta de Crecimiento de DataSciencester tiene una teoría según la cual la cantidad de tiempo que la gente se queda en el sitio está relacionada con el número de amigos que tienen en él (ella no es vicepresidenta porque sí), y quiere verificar esta afirmación.

Tras escarbar en los registros de tráfico, obtenemos una lista llamada `daily_minutes`, que muestra los minutos al día que se pasa cada usuario en DataSciencester, y la hemos ordenado de forma que sus elementos se correspondan con los elementos de nuestra lista anterior `num_friends`. Nos gustaría investigar la relación entre estas dos métricas.

Primero, veremos la covarianza, la análoga emparejada de la varianza. Mientras la varianza mide la desviación de la media de una sola variable, la covarianza mide la variación de dos variables en tándem con respecto a sus medias:

```
from scratch.linear_algebra import dot
def covariance(xs: List[float], ys: List[float]) -> float:
   assert len(xs) == len(ys), "xs and ys must have same number of elements"
   return dot(de_mean(xs), de_mean(ys)) / (len(xs)-1)
assert 22.42 < covariance(num_friends, daily_minutes) < 22.43
assert 22.42 / 60 < covariance(num_friends, daily_hours) < 22.43 / 60
```

Recordemos que `dot` suma los productos de los pares de elementos correspondientes. Cuando los elementos correspondientes de `x` e `y` están ambos por encima o por debajo de sus medias, un número positivo entra en la suma. Cuando uno está por encima de la media y el otro por debajo, es un número negativo lo que entra en la suma. De acuerdo con esto, una covarianza positiva “grande” significa que `x` tiende a ser grande cuando `y` es grande y pequeño cuando `y` es pequeño. Una covarianza negativa “grande”

significa lo contrario: que `x` tiende a ser pequeño cuando `y` es grande y viceversa. Una covarianza cercana a cero significa que no existe tal relación.

No obstante, este número puede ser difícil de interpretar por varias razones:

- Sus unidades son el producto de las unidades de las entradas (por ejemplo, amigos-minutos-al-día), lo que puede ser difícil de entender (¿qué es un "amigo-minuto-al-día"?).
- Si cada usuario tuviera el doble de amigos (pero el mismo número de minutos), la covarianza sería el doble de grande. Pero, en cierto sentido, las variables estarían igual de interrelacionadas. Dicho de otro modo, es difícil decir lo que cuenta como una covarianza "grande".

Por esta razón, es más común mirar la correlación, que divide las desviaciones estándares de ambas variables:

```
def correlation(xs: List[float], ys: List[float]) -> float:
   """Measures how much xs and ys vary in tandem about their means"""
   stdev_x = standard_deviation(xs)
   stdev_y = standard_deviation(ys)
   if stdev_x > 0 and stdev_y > 0:
      return covariance(xs, ys) / stdev_x / stdev_y
   else:
      return 0      # si no hay variación, la correlación es cero
assert 0.24 < correlation(num_friends, daily_minutes) < 0.25
assert 0.24 < correlation(num_friends, daily_hours) < 0.25
```

La `correlation` no tiene unidad y siempre está entre -1 (anticorrelación perfecta) y 1 (correlación perfecta). Un número como 0,25 representa una correlación positiva relativamente débil. Sin embargo, una cosa que olvidamos hacer fue examinar nuestros datos. Veamos la figura 5.2.

La persona que tiene 100 amigos (y que pasa únicamente 1 minuto al día en el sitio) es un enorme valor atípico, y la correlación puede ser muy sensible a estos valores. ¿Qué ocurre si le ignoramos?

```
outlier = num_friends.index(100)         # índice de valor atípico
num_friends_good = [x
             for i, x in enumerate(num_friends)
```

```
            if i != outlier]
daily_minutes_good = [x
                for i, x in enumerate(daily_minutes)
                if i != outlier]
daily_hours_good = [dm / 60 for dm in daily_minutes_good]
assert 0.57 < correlation(num_friends_good, daily_minutes_good) < 0.58
assert 0.57 < correlation(num_friends_good, daily_hours_good) < 0.58
```

Sin el valor atípico, hay una correlación mucho más fuerte (véase la figura 5.3).

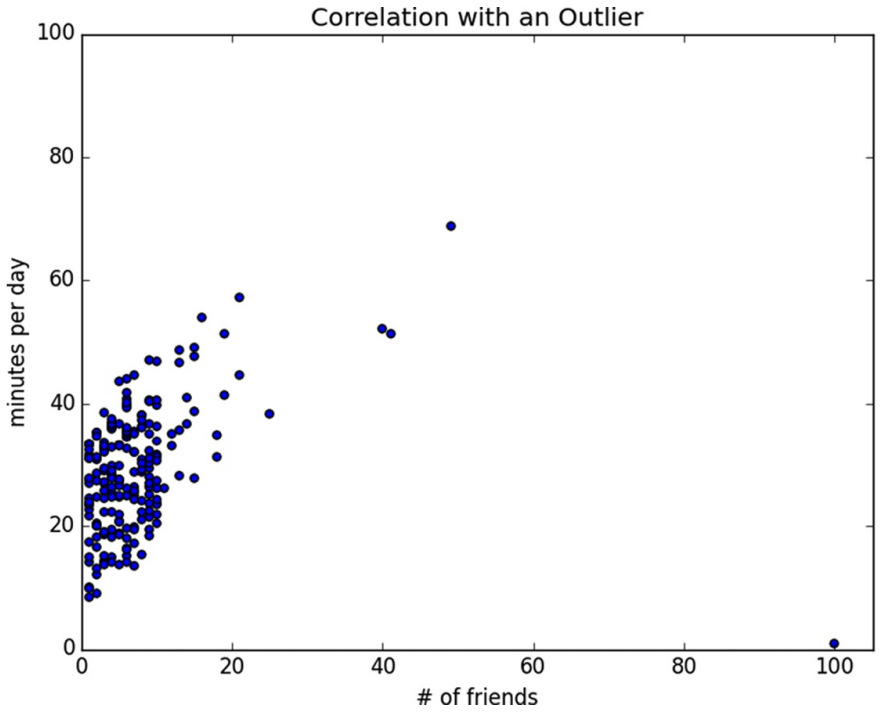


Figura 5.2. Correlación con un valor atípico.

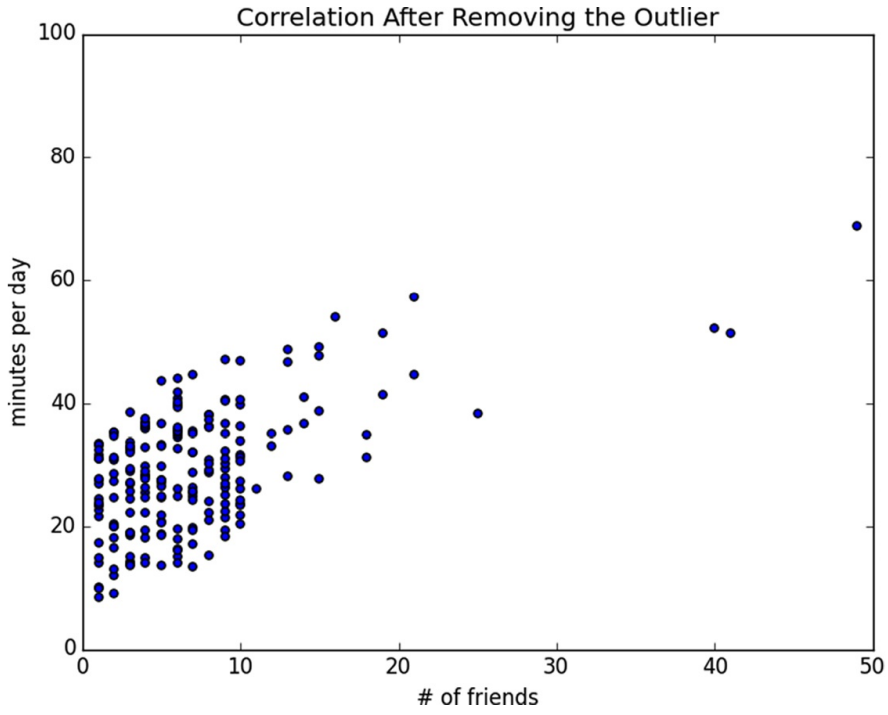


Figura 5.3. Correlación tras eliminar el valor atípico.

Seguimos investigando para descubrir que el valor atípico era realmente una cuenta de prueba interna que nadie se había preocupado nunca de eliminar. Así que su exclusión está totalmente justificada.

La paradoja de Simpson

Una sorpresa no poco habitual al analizar datos es la paradoja de Simpson, en la que las correlaciones pueden ser erróneas cuando se ignoran las variables de confusión.

Por ejemplo, imaginemos que podemos identificar todos nuestros miembros como científicos de datos de la costa este u oeste de EE. UU. Decidimos examinar los científicos de datos de qué costa son más amigables:

Costa	Nº de miembros	Nº medio de amigos
Costa oeste	101	8,2

Costa este	103	6,5

Sin duda parece que los científicos de datos de la costa oeste son más amigables que los de la costa este. Sus compañeros de trabajo avanzan todo tipo de teorías sobre la razón por la que podría ocurrir esto: ¿quizá es el sol, el café, los productos ecológicos o el ambiente relajado del Pacífico?

Pero, jugando con los datos, descubrimos algo muy extraño. Si solamente miramos las personas con doctorado, los científicos de datos de la costa este tienen más amigos en promedio.

Pero, si miramos solo las personas sin doctorado, ¡los científicos de datos de la costa este tienen también más amigos de media!

Costa	Doctorado	Nº de miembros	Nº medio de amigos
Costa oeste	Sí	35	3,1
Costa este	Sí	70	3,2
Costa oeste	No	66	10,9
Costa este	No	33	13,4

En cuanto se tienen en cuenta los doctorados de los usuarios, la correlación se va en la dirección contraria. Agrupar los datos en *buckets* como Costa Este/Costa Oeste disfrazó el hecho de que los científicos de datos de la costa este se inclinaban muchísimo más hacia los tipos con doctorado.

Este fenómeno ocurre en el mundo real con cierta regularidad. Lo esencial es que la correlación está midiendo la relación entre las dos variables, siendo todo lo demás igual. Si las clases de datos se asignan de forma aleatoria, como podría perfectamente ocurrir en un experimento bien diseñado, “siendo todo lo demás igual” podría no ser una suposición horrible.

La única forma real de evitar esto es conociendo los datos y haciendo lo posible por asegurarse de que se han revisado en busca de posibles factores de confusión. Es obvio que esto no es siempre posible. Si no tuviéramos datos sobre los logros académicos de estos 200 científicos de datos,

podríamos simplemente concluir que había algo intrínsecamente más amigable en la costa oeste.

Otras advertencias sobre la correlación

Una correlación de cero indica que no hay relación lineal entre las dos variables. Sin embargo, pueden existir otras formas de relaciones. Por ejemplo, si:

```
x = [-2, -1, 0, 1, 2]
y = [ 2, 1, 0, 1, 2]
```

Entonces `x` e `y` tienen correlación cero. Pero sin duda están relacionados: cada elemento de `y` es igual al valor absoluto del elemento correspondiente de `x`. Lo que no tienen es una relación en la que saber cómo se compara `x_i` con `mean(x)` nos ofrece información sobre cómo `y_i` se compara con `mean(y)`. Este es el tipo de relación que la correlación busca.

Además, la correlación no nos dice nada sobre lo grande que es la relación:

```
x = [-2, -1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

Las variables están perfectamente correlacionadas, pero (dependiendo de cómo estemos midiendo) es muy posible que esta relación no sea tan interesante.

Correlación y causación

Es probable que en algún momento haya oído que “correlación no es causación”, dicho lo más probable por alguien en busca de datos que plantearan un desafío a partes de su visión del mundo que era reacio a cuestionar. Sin embargo, este es un punto importante: si `x` e `y` están fuertemente correlacionados, podría significar que `x` causa `y`, que `y` causa `x`,

que cada uno causa el otro, que un tercer factor causa ambos, o nada de esto en absoluto.

Pensemos en la relación entre `num_friends` y `daily_minutes`. Es posible que tener más amigos en el sitio cause que los usuarios de DataSciencester pasen más tiempo en el sitio. Este podría ser el caso si cada amigo sube una cierta cantidad de contenido cada día, lo que significa que cuantos más amigos se tengan, más tiempo se necesita para mantenerse al día de sus actualizaciones.

Sin embargo, es también posible que, cuanto más tiempo pasen los usuarios discutiendo en foros de DataSciencester, con más frecuencia encuentren personas con su misma forma de pensar y se hagan amigos de ellas. Es decir, pasar más tiempo en el sitio causa que los usuarios tengan más amigos.

Una tercera posibilidad es que los usuarios más apasionados por la ciencia de datos pasen más tiempo en el sitio (porque lo encuentran más interesante) y consigan de forma más activa amigos de ciencia de datos (porque no quieren asociarse con ningún otro).

Una forma de sentirse más cómodos con la causalidad es realizando ensayos aleatorios. Si es posible dividir aleatoriamente los usuarios en dos grupos con demografías similares y dar a uno de los grupos una experiencia algo distinta, entonces se observa con bastante seguridad que las distintas experiencias están causando los diferentes resultados.

Por ejemplo, si no nos importara que nos acusasen con indignación de experimentar con los usuarios,[3] podríamos elegir aleatoriamente un subconjunto de usuarios y mostrarles contenido únicamente de una parte de sus amigos. Si después este subconjunto se pasara menos tiempo en el sitio, ello nos daría una cierta confianza en pensar que tener más amigos causa pasar más tiempo en el sitio.

Para saber más

- SciPy en `https://www.scipy.org`, pandas en

http://pandas.pydata.org y StatsModels en http://www.statsmodels.org, incluyen todos una gran variedad de funciones estadísticas.

- La estadística es importante. Si quiere ser un científico de datos mejor, sería una buena idea leer un libro de texto sobre estadística. En la red hay muchos disponibles, como por ejemplo:
  - *Introductory Statistics*, en https://open.umn.edu/opentextbooks/textbooks/introductorystatistics, de Douglas Shafer y Zhiyi Zhang (Saylor Foundation).
  - *OnlineStatBook*, en http://onlinestatbook.com/, de David Lane (Rice University).
  - *Introductory Statistics*, en https://openstax.org/details/introductory-statistics, de OpenStax (OpenStax College).

---

[1] http://en.wikipedia.org/wiki/Quickselect.

[2] https://es.wikipedia.org/wiki/Estimaci%C3%B3n_de_la_desviaci%C3%B3n_est%C3%A1nda

[3] https://www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html?searchResultPosition=1.

6 Probabilidad

Las leyes de la probabilidad, tan verdaderas en general, tan falaces en particular.

—Edward Gibbon

Es difícil hacer ciencia de datos sin un cierto conocimiento de la probabilidad y sus matemáticas. Al igual que ocurrió con nuestro tratamiento de la estadística en el capítulo 5, agitaremos mucho las manos y suprimiremos muchos de los tecnicismos.

Para nuestros fines lo mejor es pensar en la probabilidad como en una forma de cuantificar la incertidumbre asociada a los eventos elegidos desde un universo de eventos. En lugar de ponerse técnicos sobre lo que significan estas palabras, mejor pensemos en tirar un dado. El universo consiste en todos los resultados posibles, y cualquier subconjunto de estos resultados es un evento. Por ejemplo, "el dado saca un 1" o "el dado saca un número par".

Utilizando notación matemática, escribimos $P(E)$ para indicar "la probabilidad del evento E".

Utilizaremos la teoría de la probabilidad para crear modelos. Y para evaluar modelos. La emplearemos para todo.

Podríamos, si quisiéramos, profundizar en la filosofía del significado de la teoría de la probabilidad (lo que se haría mejor con unas cervezas). Pero no haremos eso.

Dependencia e independencia

A grandes rasgos, digamos que dos eventos E y F son dependientes si saber algo sobre si E ocurre nos da información sobre si F ocurre (y viceversa). De otro modo, son independientes.

Por ejemplo, si lanzamos una moneda dos veces, saber que el primer lanzamiento es cara no nos da información alguna sobre si en el segundo

lanzamiento saldrá también cara. Estos eventos son independientes. Por otro lado, saber si el primer lanzamiento es cara sin duda nos da información sobre si en ambos lanzamientos saldrá cruz (si en el primer lanzamiento sale cara, entonces definitivamente no es el caso de que en ambos lanzamientos salga cruz). Estos dos eventos son dependientes.

Matemáticamente, decimos que dos eventos E y F son independientes si la probabilidad de que ambos ocurran es el producto de las probabilidades de que cada uno ocurre:

$$P(E, F) = P(E)P(F)$$

En el ejemplo, la probabilidad de "primer lanzamiento cara" es de 1/2, y la probabilidad de "ambos lanzamientos cruz" es de 1/4, pero la probabilidad de "primer lanzamiento cara y ambos lanzamientos cruz" es de 0.

Probabilidad condicional

Cuando dos eventos E y F son independientes, entonces por definición tenemos:

$$P(E, F) = P(E)P(F)$$

Si no son necesariamente independientes (y si la probabilidad de F no es cero), entonces definimos la probabilidad de E "condicionada por F" como:

$$P(E|F) = P(E,F)/P(F)$$

Tendríamos que pensar en esto como la probabilidad de que E ocurra, dado que sabemos que F ocurre.

A menudo esto lo reescribimos así:

$$P(E,F) = P(E|F)P(F)$$

Cuando E y F son independientes, se puede verificar que esto da:

$$P(E|F) = P(E)$$

Que es la forma matemática de expresar que saber que F ocurrió no nos da información adicional sobre si E ocurrió.

Un ejemplo habitual y bastante complejo es el de una familia con dos hijos (desconocidos). Si suponemos que:

- Cada hijo tiene la misma probabilidad de ser niño o niña.
- El género del segundo hijo es independiente del género del primer hijo.

Entonces el evento “no niñas” tiene una probabilidad de 1/4, el evento “un niño, una niña” tiene probabilidad 1/4, y el evento “dos niñas” tiene una probabilidad también de 1/4.

Ahora podemos preguntar: ¿cuál es la probabilidad del evento “ambos hijos son niñas” (B) condicionado por el evento “el hijo mayor es una niña” (G)? Utilizando la definición de probabilidad condicional:

$$P(B|G) = P(B,G)/P(G) = P(B)/P(G) = 1/2$$

Ya que el evento B y G (“ambos hijos son niñas y el otro hijo es una niña”) es precisamente el evento B (en cuanto sabemos que ambos hijos son niñas, es necesariamente cierto que el hijo mayor sea una niña).

Lo más probable es que este resultado esté de acuerdo con su intuición.

También podríamos preguntar por la probabilidad del evento “ambos hijos son niñas” condicionado por el evento “al menos uno de los hijos es una niña” (L). Sorprendentemente, ¡la respuesta es distinta a la anterior!

Como antes, el evento B y L (“ambos hijos son niñas y al menos uno de los hijos es una niña”) es justamente el evento B. Esto significa que tenemos:

$$P(B|L) = P(B,L)/P(L) = P(B)/P(L) = 1/3$$

¿Cómo puede ser esto así? Bueno, si todo lo que sabemos es que al menos uno de los hijos es una niña, entonces es el doble de probable que la familia tenga un niño y una niña que tenga dos niñas.

Podemos comprobarlo “generando” muchas familias:

```
import enum, random
# Un Enum es un conjunto con nombre de valores enumerados.
# Podemos usarlos para que el código sea más descriptivo y legible.
class Kid(enum.Enum):
   BOY = 0
   GIRL = 1
def random_kid() -> Kid:
   return random.choice([Kid.BOY, Kid.GIRL])
both_girls = 0
older_girl = 0
either_girl = 0
random.seed(0)
for _ in range(10000):
   younger = random_kid()
   older = random_kid()
   if older == Kid.GIRL:
      older_girl += 1
   if older == Kid.GIRL and younger == Kid.GIRL:
      both_girls += 1
   if older == Kid.GIRL or younger == Kid.GIRL:
      either_girl += 1
print("P(both | older):", both_girls / older_girl)        # 0.514 ~ 1/2
print("P(both | either): ", both_girls / either_girl)     # 0.342 ~ 1/3
```

Teorema de Bayes

Uno de los mejores amigos del científico de datos es el teorema de Bayes, una forma de “revertir” las probabilidades condicionales. Digamos que necesitamos conocer la probabilidad de un cierto evento E condicionado porque algún otro evento F ocurra. Pero solamente tenemos información sobre la probabilidad de F condicionado porque E ocurra. Emplear la definición de probabilidad condicional dos veces nos dice que:

$$P(E|F) = P(E,F)/P(F) = P(F|E)P(E)/P(F)$$

El evento F puede dividirse en los dos eventos mutuamente exclusivos “F y E” y “F y no E”. Si escribimos ¬E por “no E” (es decir, “E no ocurre”),

entonces:

$P(F) = P(F,E) + P(F,\neg E)$

De modo que:

$P(E|F) = P(F|E)P(E)/[P(F|E)P(E) + P(F|\neg E)P(\neg E)]$

Que es como se suele enunciar el teorema de Bayes.

Este teorema se utiliza a menudo para demostrar por qué los científicos de datos son más inteligentes que los médicos. Imaginemos una cierta enfermedad que afecta a 1 de cada 10.000 personas. Supongamos que existe una prueba para esta enfermedad que da el resultado correcto ("enfermo" si se tiene la enfermedad y "no enfermo" si no se tiene) el 99 % de las veces.

¿Qué significa una prueba positiva? Utilicemos *T* para el evento "la prueba es positiva" y *D* para el evento "tiene la enfermedad". Entonces, el teorema de Bayes dice que la probabilidad de que tenga la enfermedad, condicionada porque la prueba sea positiva, es:

$P(D|T) = P(T|D)P(D)/[P(T|D)P(D) + P(T|\neg D)P(\neg D)]$

Aquí sabemos que $P(T|D)$, la probabilidad de que la prueba sea positiva en alguien que tenga la enfermedad, es 0,99. $P(D)$, la probabilidad de que cualquier persona tenga la enfermedad, es 1/10.000 = 0,0001. $P(T|\neg D)$, la probabilidad de que alguien que no tenga la enfermedad dé positivo en la prueba, es de 0,01. Y $P(\neg D)$, la probabilidad de que cualquier persona no tenga la enfermedad, es 0,9999. Si se sustituyen estos números en el teorema de Bayes, se obtiene:

$P(D|T) = 0{,}98\%$

Es decir, menos del 1 % de las personas cuya prueba fue positiva tienen realmente la enfermedad.

Nota: Esto supone que las personas se hacen la prueba más o menos aleatoriamente. Si solo las personas con determinados síntomas se hicieran la

prueba, en lugar de ello tendríamos que condicionar con el evento “prueba positiva y síntomas” y el número sería seguramente mucho más alto.

Una forma más intuitiva de ver esto es imaginar una población de 1 millón de personas. Podríamos esperar que 100 de ellas tuvieran la enfermedad, y que 99 de esas 100 dieran positivo. Por otro lado, supondríamos que 999.990 de ellas no tendrían la enfermedad, y que 9.999 de ellas darían positivo. Eso significa que se esperaría que solo 99 de (99 + 9.999) personas con la prueba positiva tuvieran realmente la enfermedad.

Variables aleatorias

Una variable aleatoria es una variable cuyos posibles valores tienen una distribución de probabilidad asociada. Una variable aleatoria muy sencilla es igual a 1 si al lanzar una moneda sale cara y a 0 si sale cruz. Otra más complicada mediría el número de caras que se observan al lanzar una moneda 10 veces o un valor tomado de `range(10)`, donde cada número es igualmente probable.

La distribución asociada da las probabilidades de que la variable realice cada uno de sus posibles valores. La variable lanzamiento de moneda es igual a 0 con una probabilidad de 0,5 y a 1 con una probabilidad de 0,5. La variable `range(10)` tiene una distribución que asigna una probabilidad de 0,1 a cada uno de los números de 0 a 9.

En ocasiones, hablaremos del valor esperado de una variable aleatoria, que es la media de sus valores ponderados por sus probabilidades. La variable lanzamiento de moneda tiene un valor esperado de 1/2 (= 0 * 1/2 + 1 * 1/2), y la variable `range(10)` tiene un valor esperado de 4,5.

Las variables aleatorias pueden estar condicionadas por eventos igual que el resto de eventos puede estarlo. Volviendo al ejemplo de los dos hijos de la sección “Probabilidad condicional”, si X es la variable aleatoria que representa el número de niñas, X es igual a 0 con una probabilidad de 1/4, 1 con una probabilidad de 1/2 y 2 con una probabilidad de 1/4.

Podemos definir una nueva variable aleatoria *Y* que da el número de niñas condicionado por al menos que uno de los hijos sea una niña. Entonces *Y* es igual a 1 con una probabilidad de 2/3 y a 2 con una probabilidad de 1/3. Y una variable *Z* que es el número de niñas condicionado porque el otro hijo sea una niña es igual a 1 con una probabilidad de 1/2 y a 2 con una probabilidad de 1/2.

La mayor parte de las veces estaremos utilizando variables aleatorias de forma implícita en lo que hagamos sin atraer especialmente la atención hacia ellas. Pero si mira atentamente las verá.

Distribuciones continuas

El lanzamiento de una moneda se corresponde con una distribución discreta, que asocia probabilidad positiva con resultados discretos. A menudo querremos modelar distribuciones a lo largo de una serie de resultados (para nuestros fines, estos resultados siempre serán números reales, aunque ese no sea siempre el caso en la vida real). Por ejemplo, la distribución uniforme pone el mismo peso en todos los números entre 0 y 1.

Como hay infinitos números entre 0 y 1, eso significa que el peso que asigna a puntos individuales debe ser necesariamente 0. Por esta razón representamos una distribución continua con una función de densidad de probabilidad PDF (*Probability Density Function*) tal que la probabilidad de ver un valor en un determinado intervalo es igual a la integral de la función de densidad sobre el intervalo.

> **Nota:** Si tiene un poco oxidado el cálculo de integrales, una forma más sencilla de comprender esto es que si una distribución tiene la función de densidad f, entonces la probabilidad de ver un valor entre x y $x + h$ es aproximadamente de $h * f(x)$ si h es pequeño.

La función de densidad para la distribución uniforme es sencillamente:

```
def uniform_pdf(x: float) -> float:
```

```
    return 1 if 0 <= x < 1 else 0
```

La probabilidad de que una variable aleatoria siguiendo esa distribución esté entre 0,2 y 0,3 es de 1/10, como era de esperar. La variable `random.random` de Python es (pseudo)aleatoria con una densidad uniforme.

Con frecuencia estaremos más interesados en la función de distribución acumulativa CDF (*Cumulative Distribution Function*), que da la probabilidad de que una variable aleatoria sea menor o igual a un determinado valor. No es difícil crear la función CDF para la distribución uniforme (véase la figura 6.1):

```
def uniform_cdf(x: float) -> float:
    """Returns the probability that a uniform random variable is <= x"""
    if x < 0: return 0      # aleatoria uniforme nunca es menor que 0
    elif x < 1: return x    # p.ej. P(X <= 0.4) = 0.4
    else: return 1          # aleatoria uniforme es siempre menor que 1
```

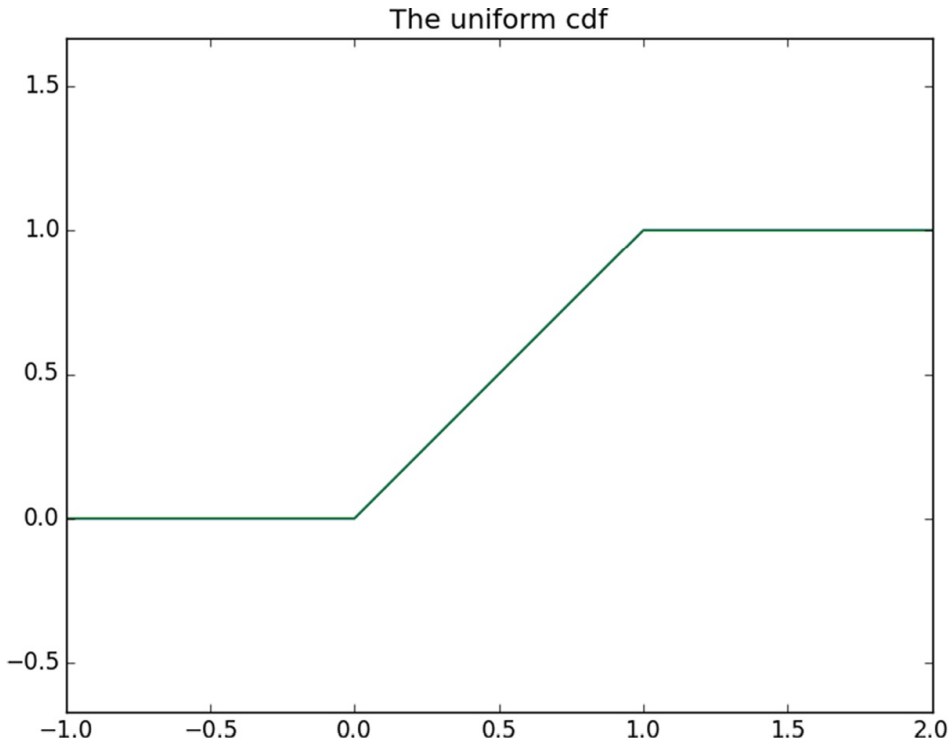


Figura 6.1. La función CDF uniforme.

La distribución normal

La distribución normal es la distribución clásica en forma de campana y se determina completamente con dos parámetros: su media μ (mu) y su desviación estándar σ (sigma). La media indica dónde está centrada la campana, y la desviación estándar lo "ancha" que es.

Tiene la función PDF:

$$f(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Que podemos implementar como:

```
import math
SQRT_TWO_PI = math.sqrt(2 * math.pi)
def normal_pdf(x: float, mu: float = 0, sigma: float = 1) -> float:
   return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (SQRT_TWO_PI * sigma))
```

En la figura 6.2 trazamos algunas de estas funciones PDF para ver cómo quedan:

```
import matplotlib.pyplot as plt
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'—',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```

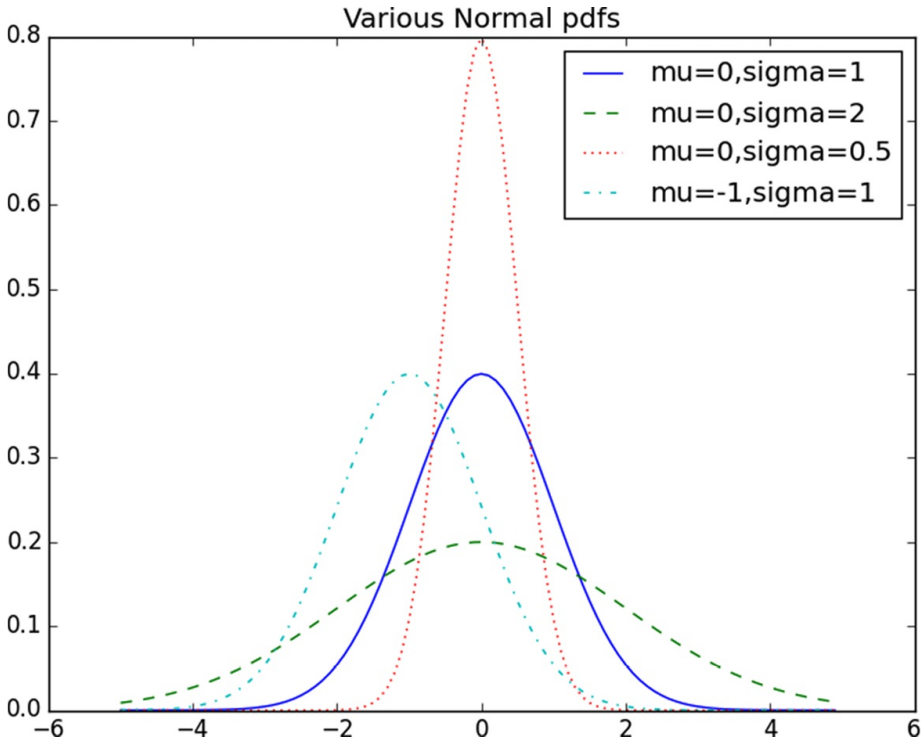


Figura 6.2. Varias funciones PDF normales.

Cuando $\mu = 0$ y $\sigma = 1$, se denomina distribución normal estándar. Si Z es una variable aleatoria normal estándar, entonces resulta que:

$$X = \sigma Z + \mu$$

También es normal, pero con media μ y desviación estándar σ. A la inversa, si X es una variable aleatoria normal con media μ y desviación estándar σ:

$$Z = (X - \mu)/\sigma$$

Es una variable normal estándar.

La función CDF para la distribución normal no se puede escribir de una forma "elemental", pero podemos hacerlo utilizando la función de error `math.erf` de Python:[1]

```
def normal_cdf(x: float, mu: float = 0, sigma: float = 1) -> float:
return (1 + math.erf((x–mu) / math.sqrt(2) / sigma)) / 2
```

De nuevo, en la figura 6.3 trazamos algunas CDF:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'—',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```

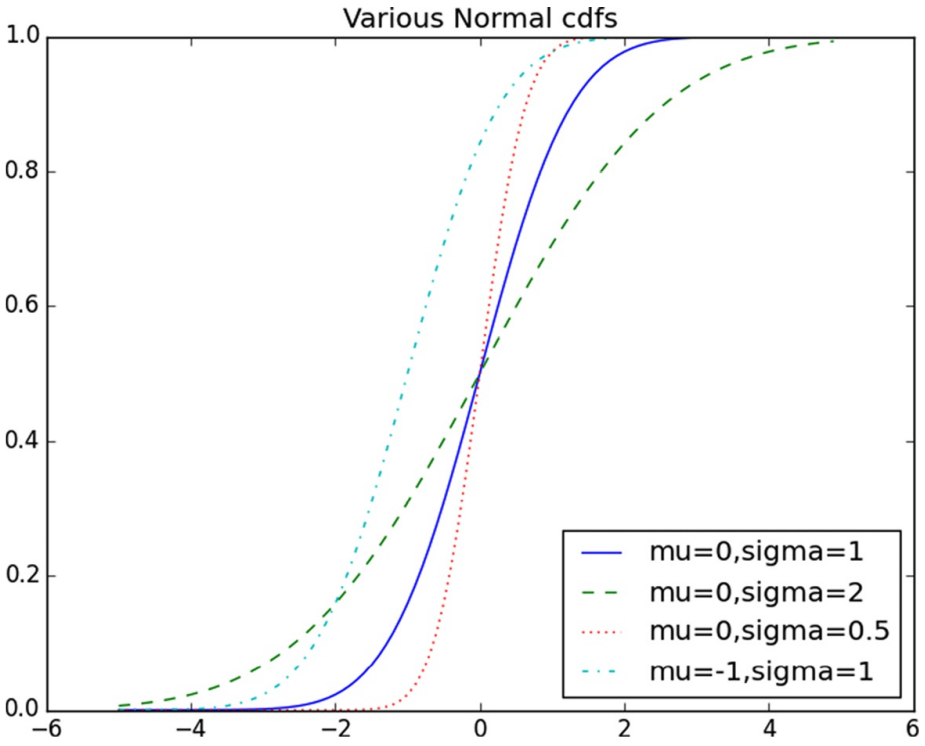


Figura 6.3. Varias CDF normales.

Algunas veces tendremos que invertir `normal_cdf` para encontrar el valor correspondiente a una determinada probabilidad. No hay una forma sencilla de calcular su inverso, pero `normal_cdf` es continuo y estrictamente creciente, de modo que utilizaremos una búsqueda binaria:[2]

```
def inverse_normal_cdf(p: float,
                       mu: float = 0,
                       sigma: float = 1,
                       tolerance: float = 0.00001) -> float:
    """Find approximate inverse using binary search"""
    # si no es estándar, calcula estándar y redimensiona
```

```
if mu != 0 or sigma != 1:
   return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)
low_z = -10.0                    # normal_cdf(-10) es (muy cercano a) 0
hi_z = 10.0                      # normal_cdf(10) es (muy cercano a) 1
while hi_z–low_z > tolerance:
   mid_z = (low_z + hi_z) /      # Considera el punto medio
   2
   mid_p =                       # y el valor de la CDF allí
   normal_cdf(mid_z)
   if mid_p < p:
      low_z = mid_z              # Punto medio demasiado bajo, busca por
                                 encima
   else:
      hi_z = mid_z               # Punto medio demasiado alto, busca por
                                 debajo
return mid_z
```

La función bisecciona repetidamente intervalos hasta que se estrecha en una *Z* que esté lo bastante cerca de la probabilidad deseada.

El teorema central del límite

Una razón por la que la distribución normal es tan útil es el teorema central del límite, que dice (básicamente) que una variable aleatoria, definida como la media de un gran número de variables aleatorias independientes y distribuidas de manera idéntica, está en sí misma aproximadamente y normalmente distribuida.

En particular, si *x*1, ..., *x*n son variables aleatorias con media μ y desviación estándar σ, y si *n* es grande, entonces:

$$\frac{1}{n}(x_1 + ... + x_n)$$

Está aproximadamente y normalmente distribuida con media μ y desviación estándar

$$\sigma/\sqrt{n}$$

. De forma equivalente (pero a menudo más útil):

$$\frac{\left(x_1 + ... + x_n\right) - \mu n}{\sigma\sqrt{n}}$$

Está aproximadamente y normalmente distribuida con media 0 y desviación estándar 1.

Una forma sencilla de ilustrar esto es mirando las variables aleatorias binomiales, que tienen dos parámetros *n* y *p*. Una variable aleatoria Binomial(*n*,*p*) no es más que la suma de *n* variables aleatorias independientes Bernoulli(*p*), cada una de las cuales es igual a 1 con una probabilidad de *p* y a 0 con una probabilidad 1 - *p*:

```
def bernoulli_trial(p: float) -> int:
   """Returns 1 with probability p and 0 with probability 1-p"""
   return 1 if random.random() < p else 0
def binomial(n: int, p: float) -> int:
   """Returns the sum of n bernoulli(p) trials"""
   return sum(bernoulli_trial(p) for _ in range(n))
```

La media de una variable Bernoulli(*p*) es *p*, y su desviación estándar es $\sqrt{p(1-p)}$. El teorema central del límite dice que cuando *n* es más grande, una variable Binomial(*n*,*p*) es aproximadamente una variable aleatoria normal con media $\mu = np$ y desviación estándar $\sigma = \sqrt{np(1-p)}$.

Si trazamos ambas, se puede ver fácilmente el parecido:

```
from collections import Counter
def binomial_histogram(p: float, n: int, num_points: int) -> None:
   """Picks points from a Binomial(n, p) and plots their histogram"""
   data = [binomial(n, p) for _ in range(num_points)]
   # usa un gráfico de barras para mostrar las muestras binomiales reales
   histogram = Counter(data)
   plt.bar([x–0.4 for x in histogram.keys()],
        [v / num_points for v in histogram.values()],
        0.8,
        color='0.75')
   mu = p * n
   sigma = math.sqrt(n * p * (1–p))
   # usa un gráfico de líneas para mostrar la aproximación normal
   xs = range(min(data), max(data) + 1)
   ys = [normal_cdf(i + 0.5, mu, sigma)–normal_cdf(i–0.5, mu, sigma)
      for i in xs]
```

```
    plt.plot(xs,ys)
    plt.title("Binomial Distribution vs. Normal Approximation")
    plt.show()
```

Por ejemplo, cuando llamamos a `make_hist(0.75, 100, 10000)`, obtenemos el gráfico de la figura 6.4.

La moraleja de esta aproximación es que, si queremos conocer la probabilidad de que (supongamos) al lanzar una moneda se saquen más de 60 caras en 100 lanzamientos, se puede estimar como la probabilidad de que un Normal(50,5) es mayor que 60, que es más fácil que calcular la función CDF Binomial(100,0.5) (aunque en la mayoría de las aplicaciones probablemente utilizaríamos software estadístico que calcularía felizmente cualesquiera probabilidades deseadas).

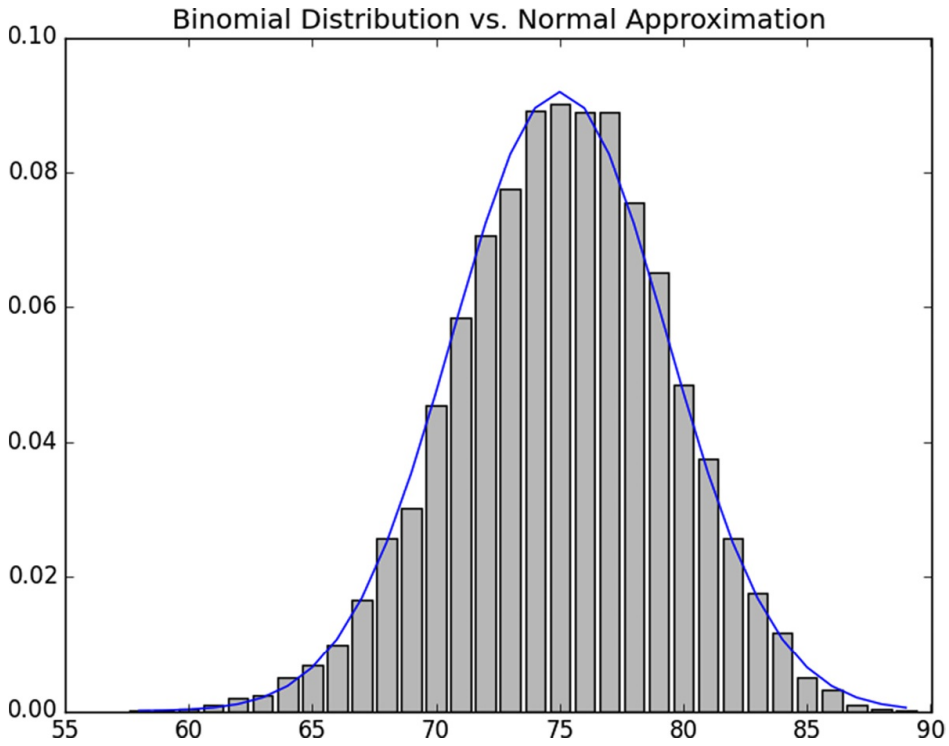


Figura 6.4. El resultado de `binomial_histogram`.

Para saber más

- scipy.stats, en https://docs.scipy.org/doc/scipy/reference/stats.html, contiene funciones PDF y CDF para la mayoría de las distribuciones de probabilidad más conocidas.
- ¿Recuerda que, al final del capítulo 5, dije que sería una buena idea estudiar un libro de texto de estadística? Pues también lo sería estudiarlo de probabilidad. El mejor que conozco que está disponible en la red es *Introduction to ProbabilityK*, en https://math.dartmouth.edu/~prob/prob/prob.pdf, de Charles M. Grinstead y J. Laurie Snell (American Mathematical Society).

---

[1] https://es.wikipedia.org/wiki/Funci%C3%B3n_error.

[2] https://es.wikipedia.org/wiki/B%C3%BAsqueda_binaria.

7 Hipótesis e inferencia

Es signo de una persona realmente inteligente el sentirse conmovida por las estadísticas.

—George Bernard Shaw

¿Qué haremos con toda esta estadística y toda esta teoría de la probabilidad? La parte científica de la ciencia de datos implica habitualmente la formación y comprobación de hipótesis sobre nuestros datos y sobre los procesos que los generan.

Comprobación de hipótesis estadísticas

A menudo, como científicos de datos, querremos probar si una determinada hipótesis es probable que sea cierta. Para nuestros fines, las hipótesis son aseveraciones como “esta moneda está equilibrada”, o “los científicos de datos prefieren Python a R”, o “es más probable que la gente salga de la página sin haber leído el contenido si aparece un irritante anuncio intercalado con un botón de cerrar diminuto y difícil de encontrar”, que se pueden traducir en estadísticas sobre datos. Bajo diferentes supuestos, se pueden considerar esas estadísticas como observaciones de variables aleatorias de distribuciones conocidas, lo que nos permite hacer afirmaciones sobre la probabilidad de que esos supuestos se cumplan.

En una configuración clásica, tenemos una hipótesis nula, H_0, que representa una cierta posición predeterminada, y otra hipótesis alternativa, H_1, con la que nos gustaría compararla. Utilizamos la estadística para decidir si podemos rechazar H_0 como falsa o no. Probablemente esto tiene más sentido con un ejemplo.

Ejemplo: Lanzar una moneda

Imaginemos que tenemos una moneda y queremos comprobar si es justa. Haremos la suposición de que la moneda tiene una cierta probabilidad *p* de sacar cara, de modo que nuestra hipótesis nula es que la moneda es justa (es decir, que $p = 0{,}5$). Comprobaremos esto frente a la hipótesis alternativa $p \neq 0{,}5$.

En particular, nuestra prueba implicará lanzar la moneda un número *n* de veces y contar el número de caras que salen, *X*. Cada lanzamiento de la moneda es un ensayo de Bernoulli, lo que significa que *X* es una variable aleatoria Binomial(*n*,*p*), la cual (como ya vimos en el capítulo 6) podemos aproximar utilizando la distribución normal:

```
from typing import Tuple
import math
def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
   """Returns mu and sigma corresponding to a Binomial(n, p)"""
   mu = p * n
   sigma = math.sqrt(p * (1-p) * n)
   return mu, sigma
```

Siempre que una variable aleatoria siga una distribución normal, podemos utilizar `normal_cdf` para averiguar la probabilidad de que su valor realizado esté dentro o fuera de un determinado intervalo:

```
from scratch.probability import normal_cdf
# La normal cdf _es_ la probabilidad de que la variable esté por debajo de un
límite
normal_probability_below = normal_cdf
# Está por encima del límite si no está por debajo
def normal_probability_above(lo: float,
 mu: float = 0,
 sigma: float = 1) -> float:
   """The probability that an N(mu, sigma) is greater than lo."""
   return 1-normal_cdf(lo, mu, sigma)
# Está en medio si es menor que hi, pero no menor que lo
def normal_probability_between(lo: float,
 hi: float,
 mu: float = 0,
 sigma: float = 1) -> float:
```

```
  """The probability that an N(mu, sigma) is between lo and hi."""
  return normal_cdf(hi, mu, sigma)–normal_cdf(lo, mu, sigma)
# Está fuera si no está en medio
def normal_probability_outside(lo: float,
 hi: float,
 mu: float = 0,
 sigma: float = 1) -> float:
  """The probability that an N(mu, sigma) is not between lo and hi."""
  return 1–normal_probability_between(lo, hi, mu, sigma)
```

También podemos hacer lo contrario: encontrar o bien la región que no esté en un extremo o el intervalo (simétrico) en torno a la media que se tiene en cuenta para un determinado nivel de probabilidad. Por ejemplo, si queremos encontrar un intervalo centrado en la media y que contenga un 60 % de probabilidad, entonces tenemos que hallar los límites en los que los extremos superior e inferior contienen cada uno un 20 % de la probabilidad (dejando el 60 %):

```
from scratch.probability import inverse_normal_cdf
def normal_upper_bound(probability: float,
               mu: float = 0,
               sigma: float = 1) -> float:
  """Returns the z for which P(Z <= z) = probability"""
  return inverse_normal_cdf(probability, mu, sigma)
def normal_lower_bound(probability: float,
               mu: float = 0,
               sigma: float = 1) -> float:
  """Returns the z for which P(Z >= z) = probability"""
  return inverse_normal_cdf(1–probability, mu, sigma)
def normal_two_sided_bounds(probability: float,
                 mu: float = 0,
                 sigma: float = 1) -> Tuple[float, float]:
  """
  Returns the symmetric (about the mean) bounds
  that contain the specified probability
  """
  tail_probability = (1–probability) / 2
  # el extremo superior tendría tail_probability por encima
  upper_bound = normal_lower_bound(tail_probability, mu, sigma)
  # el extremo inferior tendría tail_probability por debajo
  lower_bound = normal_upper_bound(tail_probability, mu, sigma)
  return lower_bound, upper_bound
```

En particular, digamos que elegimos lanzar la moneda n = 1.000 veces. Si nuestra hipótesis nula es cierta, X debería estar distribuida aproximadamente normal con una media de 500 y una desviación estándar de 15,8:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

Tenemos que tomar una decisión sobre la significancia (lo dispuestos que estamos a cometer un error tipo 1 o "falso positivo", en el que rechazamos H_0 incluso aunque sea verdadero). Por razones perdidas en los anales de la historia, esta voluntad se suele establecer en un 5 % o en un 1 %. Elijamos un 5 %.

Consideremos la prueba que rechaza H_0 si X queda fuera de los extremos dados por:

```
# (469, 531)
lower_bound, upper_bound = normal_two_sided_bounds(0.95, mu_0, sigma_0)
```

Suponiendo que p es de verdad igual a 0,5 (es decir, H_0 es verdadero), solamente hay un 5 % de posibilidades de que observemos una X que esté fuera de este intervalo, que es exactamente la significancia que queríamos. Dicho de otro modo, si H_0 es verdadero, entonces aproximadamente 19 veces de 20 esta prueba dará el resultado correcto.

También nos interesaremos a menudo por la potencia de una prueba de hipótesis, que es la probabilidad de no cometer un error tipo 2 ("falso negativo"), en el que no rechazamos H_0 incluso aunque sea falso. Para medir esto, tenemos que especificar lo que significa exactamente que H_0 sea falso (saber simplemente que p no es 0,5 no nos da mucha información sobre la distribución de X). En particular, comprobemos lo que ocurre si p es realmente 0,55, o sea, que la moneda tiende levemente hacia la cara.

En ese caso, podemos calcular la potencia de la prueba con:

```
# extremos en 95 % basados en suponer que p es 0,5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)
# mu y sigma reales basadas en p = 0,55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)
# un error tipo 2 significa que no rechazamos la hipótesis nula
# lo que ocurrirá cuando X siga en nuestro intervalo original
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1-type_2_probability                          # 0.887
```

Imaginemos, en lugar de esto, que nuestra hipótesis nula fuera que la moneda no estuviera inclinada hacia la cara, o que $p \leq 0{,}5$. En ese caso, queremos una prueba de una sola cara, que rechace la hipótesis nula cuando X es mucho mayor que 500, pero no cuando X es menor que 500. Así, una prueba de significancia del 5 % implica utilizar `normal_probability_below` para encontrar el límite bajo el que se sitúa el 95 % de la probabilidad:

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# es 526 (<531, ya que necesitamos más probabilidad en el límite superior)
type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1–type_2_probability                              # 0.936
```

Esta es una comprobación más potente, dado que ya no rechaza H_0 cuando X está por debajo de 469 (que es muy improbable que ocurra si H_1 es verdadero), y en su lugar rechaza H_0 cuando X está entre 526 y 531 (lo que tiene alguna probabilidad de ocurrir si H_1 es verdadero).

Valores p

Una forma distinta de pensar en la prueba anterior involucra a los valores *p* o *p-values*. En vez de elegir límites basándonos en un tope de probabilidad, calculamos la probabilidad (suponiendo que H_0 sea verdadero) de ver un valor al menos tan extremo como el que realmente observamos.

Para nuestra prueba de dos caras de si la moneda es justa, hacemos estos cálculos:

```
def two_sided_p_value(x: float, mu: float = 0, sigma: float = 1) -> float:
   """
   How likely are we to see a value at least as extreme as x (in either
   direction) if our values are from an N(mu, sigma)?
   """
   if x >= mu:
      # x es mayor que la media, así el extremo es todo lo que es mayor que x
      return 2 * normal_probability_above(x, mu, sigma)
   else:
```

```
    # x es menor que la media, así el extremo es todo lo que es menor que x
    return 2 * normal_probability_below(x, mu, sigma)
```

Si viéramos 530 caras, este sería el cálculo:

```
two_sided_p_value(529.5, mu_0, sigma_0) # 0.062
```

Nota: ¿Por qué hemos utilizado un valor de `529.5` en lugar de utilizar `530`? Esto es lo que se llama corrección por continuidad.[1] Refleja el hecho de que `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` es una mejor estimación de la probabilidad de ver 530 caras que `normal_probability_between(530, 531, mu_0, sigma_0)`.
En consecuencia, `normal_probability_above(529.5, mu_0, sigma_0)` es una mejor estimación de la probabilidad de ver al menos 530 caras. Quizá se haya dado cuenta de que también hemos utilizado esto en el código que producía la figura 6.4.

Una forma de autoconvencernos de que es una estimación razonable es utilizando una simulación:

```
import random
extreme_value_count = 0
for _ in range(1000):
   num_heads = sum(1 if random.random() < 0.5        # Cuenta el nº de caras
   else 0
              for _ in range(1000))                  # en 1.000 lanzamientos,
   if num_heads >= 530 or num_heads <= 470:          # y cuenta con qué
                                                     frecuencia
      extreme_value_count += 1                       # el nº es ‘extremo’
# el p-value era 0,062 => ~62 valores extremos de 1.000
assert 59 < extreme_value_count < 65, f"{extreme_value_count}"
```

Como el valor *p* es mayor que nuestra significancia del 5 %, no rechazamos la hipótesis nula. Si viéramos sin embargo 532 caras, el valor *p* sería:

```
two_sided_p_value(531.5, mu_0, sigma_0)      # 0,0463
```

Que es menor que la significancia del 5 %, lo que indica que sí la rechazaríamos. Es exactamente la misma prueba que antes, solo que con una

forma diferente de enfocar las estadísticas.

De forma similar, tendríamos:

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

Para nuestra prueba de una sola cara, si viéramos 525 caras calcularíamos:

```
upper_p_value(524.5, mu_0, sigma_0)     # 0,061
```

Lo que significa que no rechazaríamos la hipótesis nula. Si viéramos 527 caras, el cálculo sería:

```
upper_p_value(526.5, mu_0, sigma_0)     # 0,047
```

Y sí rechazaríamos la hipótesis nula.

Advertencia: Asegúrese de que sus datos están más o menos normalmente distribuidos antes de utilizar `normal_probability_above` para calcular valores *p*. Los anales de la ciencia de datos mal practicada están llenos de ejemplos de personas opinando que la posibilidad de que algún evento observado se produzca aleatoriamente es una entre un millón, cuando lo que realmente quieren decir es "la posibilidad, suponiendo que los datos estén normalmente distribuidos", lo que no tiene mucho sentido si los datos no lo están.
Existen diferentes pruebas estadísticas para la normalidad, pero incluso trazar los datos es un buen comienzo.

Intervalos de confianza

Hemos estado probando hipótesis sobre el valor de la probabilidad *p* de que salga cara, que es un parámetro de la distribución desconocida "cara". Cuando es este el caso, un tercer método es construir un intervalo de confianza en torno al valor observado del parámetro.

Por ejemplo, podemos estimar la probabilidad de la moneda injusta mirando el valor medio de las variables de Bernoulli correspondientes a cada lanzamiento (1 si es cara, 0 si es cruz). Si observamos 525 caras en 1.000

lanzamientos, entonces estimamos que *p* es igual a 0,525.

¿Qué confianza podemos tener en esta estimación? Bueno, si conociéramos el valor exacto de *p*, el teorema central del límite (recuerde la sección del mismo nombre del capítulo 6) nos dice que la media de dichas variables de Bernoulli debería ser aproximadamente normal, con una media de *p* y una desviación estándar de:

```
math.sqrt(p * (1–p) / 1000)
```

Aquí no conocemos *p*, así que por eso utilizamos nuestra estimación:

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1–p_hat) / 1000)      # 0,0158
```

Esto no está del todo justificado, pero la gente parece hacerlo de todas formas. Utilizando la aproximación normal, concluimos que "tenemos una confianza del 95 %" en que el siguiente intervalo contiene el verdadero parámetro *p*:

```
normal_two_sided_bounds(0.95, mu, sigma)      # [0.4940, 0.5560]
```

Nota: Esta es una afirmación sobre el intervalo, no sobre *p*. Se debería entender como la aseveración de que, si repitiéramos el experimento muchas veces, el 95 % del tiempo el parámetro "verdadero" (que es el mismo cada vez) estaría dentro del intervalo de confianza observado (que podría ser diferente cada vez).

En particular, no concluimos que la moneda sea injusta, ya que 0,5 está dentro de nuestro intervalo de confianza.

Si lo que viéramos fueran 540 caras, entonces tendríamos:

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1–p_hat) / 1000)      # 0,0158
normal_two_sided_bounds(0.95, mu, sigma)         # [0.5091, 0.5709]
```

Aquí, "moneda justa" no está en el intervalo de confianza (la hipótesis de

“moneda justa” no pasa una prueba que se supone que debiera pasar el 95 % de las veces si fuera verdadera).

p-hacking o dragado de datos

Un procedimiento que rechace erróneamente la hipótesis nula solo el 5 % de las veces rechazará (por definición) erróneamente la hipótesis nula el 5 % de las veces:

```
from typing import List
def run_experiment() -> List[bool]:
   """Flips a fair coin 1000 times, True = heads, False = tails"""
   return [random.random() < 0.5 for _ in range(1000)]
def reject_fairness(experiment: List[bool]) -> bool:
   """Using the 5% significance levels"""
   num_heads = len([flip for flip in experiment if flip])
   return num_heads < 469 or num_heads > 531
random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                for experiment in experiments
                if reject_fairness(experiment)])
assert num_rejections == 46
```

Lo que esto significa es que, si pretendemos encontrar resultados “significativos”, es posible hallarlos. Probando las hipótesis suficientes sobre nuestro conjunto de datos, casi con seguridad una de ellas será significativa. Si eliminamos los valores atípicos correctos, probablemente podremos obtener el valor p inferior a 0,05 (hemos hecho algo vagamente parecido en la sección “Correlación” del capítulo 5; ¿se había dado cuenta?).

Esto a veces se denomina *p-hacking* o dragado de datos,[2] y en algunos aspectos es una consecuencia de la “infraestructura de inferencia de valores *p*”. Un buen artículo que critica este enfoque es “The Earth Is Round”[3] de Jacob Cohen.

Si queremos hacer buena ciencia, deberemos determinar nuestras hipótesis antes de mirar los datos, limpiar los datos sin tener en mente las hipótesis y

recordar que los valores *p* no son sustitutos del sentido común (un método alternativo se trata en la sección “Inferencia bayesiana”, antes de finalizar este capítulo).

Ejemplo: Realizar una prueba A/B

Una de sus principales responsabilidades en DataSciencester es la optimización de la experiencia, un eufemismo para intentar que la gente haga clic en los anuncios. Uno de los anunciantes ha desarrollado una nueva bebida energética destinada a los científicos de datos, y el vicepresidente de Publicidad quiere que elija entre el anuncio A (“¡menudo sabor!) y el anuncio B (“¡menos prejuicios!”).

Como somos científicos, decidimos realizar un experimento mostrando de forma aleatoria a los visitantes del sitio uno de los dos anuncios y haciendo un seguimiento de cuántos de ellos hacen clic sobre cada uno.

Si 990 de 1.000 espectadores del anuncio A hacen clic en él, mientras que solamente 10 de 1.000 del anuncio B hacen clic en el B, podríamos estar bastante seguros de que A es el mejor anuncio. Pero ¿qué pasa si las diferencias no son tan abismales? Aquí es cuando entra en juego la inferencia estadística.

Digamos que *NA* personas ven el anuncio A y que *nA* de ellas hacen clic en él. Podemos pensar en cada visión del anuncio como en un ensayo de Bernoulli, donde *pA* es la probabilidad de que alguien haga clic en el anuncio A. Entonces (si *NA* es grande, que lo es aquí) sabemos que *nA*/*NA* es aproximadamente una variable aleatoria normal, con una media de *pA* y una desviación estándar de $\sigma_A = \sqrt{p_A(1 - p_A)/N_A}$.

De forma similar, *nB*/*NB* es aproximadamente una variable aleatoria normal con una media de *pB* y una desviación estándar de $\sigma_B = \sqrt{p_B(1 - p_B)/N_B}$. Podemos expresar esto en código del siguiente modo:

```
def estimated_parameters(N: int, n: int) -> Tuple[float, float]:
   p = n / N
   sigma = math.sqrt(p * (1–p) / N)
   return p, sigma
```

Si suponemos que esas dos normales son independientes (lo que parece razonable, ya que los ensayos de Bernoulli individuales podrían serlo), entonces su diferencia también debería ser una normal con media *PB–PA* y desviación estándar $\sqrt{\sigma_A^2 + \sigma_B^2}$.

> **Nota:** Esto es un poco trampa. Las mates solo cuadran exactamente así si conocemos las desviaciones estándar. Aquí estamos estimándolas desde los datos, lo que significa que en realidad deberíamos estar utilizando una distribución *t*. Pero, para conjuntos de datos bastante grandes, se aproxima lo suficiente como para no suponer una gran diferencia.

Esto significa que podemos probar la hipótesis nula de que *pA* y *pB* son iguales (es decir, que *pA* - *pB* = 0) utilizando la estadística:

```
def a_b_test_statistic(N_A: int, n_A: int, N_B: int, n_B: int) -> float:
   p_A, sigma_A = estimated_parameters(N_A, n_A)
   p_B, sigma_B = estimated_parameters(N_B, n_B)
   return (p_B–p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

Que debería ser aproximadamente una normal estándar.

Por ejemplo, si “menudo sabor” obtiene 200 clics de 1.000 visitantes y “menos prejuicios” obtiene 180 clics de 1.000 espectadores, la estadística es igual a:

```
z = a_b_test_statistic(1000, 200, 1000, 180)      # -1,14
```

La probabilidad de ver una diferencia tan grande si las medias fueran realmente iguales sería:

```
two_sided_p_value(z)      # 0,254
```

Que es tan grande que no podemos concluir que haya mucha diferencia. Por otro lado, si “menos prejuicios” solo obtuviera 150 clics, tendríamos:

```
z = a_b_test_statistic(1000, 200, 1000, 150)      # -2,94
two_sided_p_value(z)                              # 0,003
```

Lo que significa que solamente hay una probabilidad de 0,003 de que veamos una diferencia tan grande si los anuncios fueran igualmente efectivos.

Inferencia bayesiana

Los procedimientos que hemos visto han supuesto realizar afirmaciones de probabilidad sobre nuestras pruebas: por ejemplo, “solo hay un 3 % de posibilidades de que observemos una estadística tan extrema si nuestras hipótesis nulas fueran verdaderas”.

Un método distinto a la inferencia implica tratar los parámetros desconocidos como variables aleatorias. El analista (que es usted) empieza con una distribución previa para los parámetros y después utiliza los datos observados y el teorema de Bayes para obtener una distribución posterior actualizada para los parámetros. En lugar de hacer juicios de probabilidad sobre las pruebas, mejor hacer juicios de probabilidad sobre los parámetros.

Por ejemplo, cuando el parámetro desconocido es una probabilidad (como en nuestro ejemplo del lanzamiento de la moneda), a menudo empleamos una previa de la distribución beta, que coloca toda su probabilidad entre 0 y 1:

```
def B(alpha: float, beta: float) -> float:
   """A normalizing constant so that the total probability is 1"""
   return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)
def beta_pdf(x: float, alpha: float, beta: float) -> float:
   if x <= 0 or x >= 1:           # no hay peso fuera de [0, 1]
      return 0
   return x ** (alpha–1) * (1–x) ** (beta–1) / B(alpha, beta)
```

Por lo general, esta distribución centra su peso en:

```
alpha / (alpha + beta)
```

Y, cuanto más grandes sean `alpha` y `beta`, más “ajustada” es la distribución.

Por ejemplo, si `alpha` y `beta` son ambas 1, es la distribución uniforme como tal (centrada en 0,5, muy dispersa). Si `alpha` es mucho mayor que

`beta`, la mayor parte del peso está cerca de 1. Y si `alpha` es mucho menor que `beta`, la mayor parte del peso está cerca de 0. La figura 7.1 muestra varias distribuciones beta diferentes.

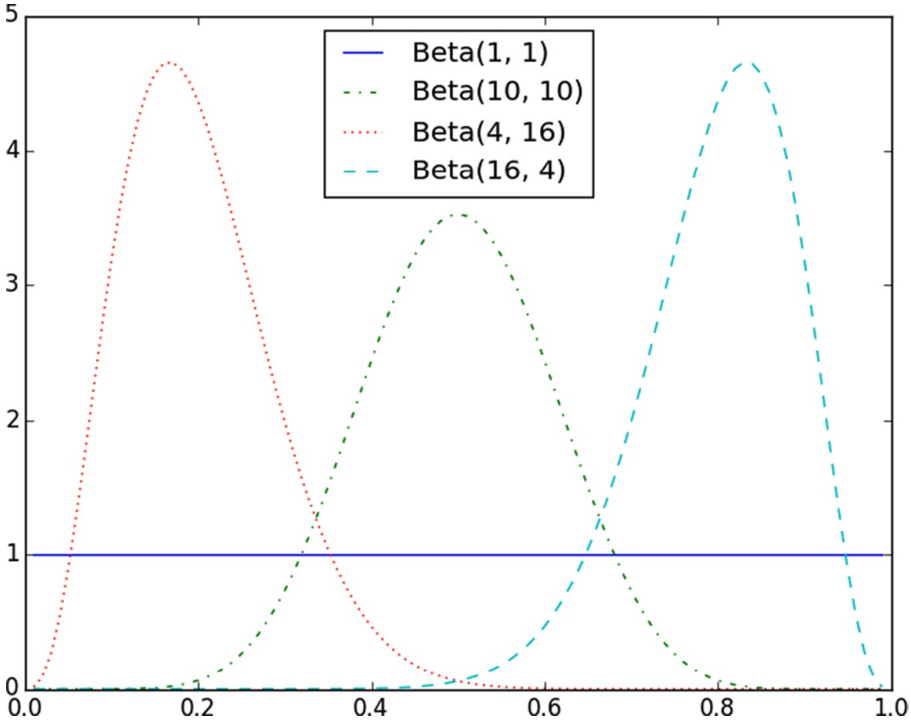


Figura 7.1. Distribuciones beta de ejemplo.

Digamos que suponemos una distribución anterior en *p*. Quizá no queramos tomar partido sobre si la moneda es justa, así que elegimos que `alpha` y `beta` sean ambas 1. O quizá tenemos clarísimo que la moneda saca cara el 55 % de las veces, por lo que elegimos que `alpha` sea igual a 55 y `beta` sea igual a 45.

Entonces lanzamos nuestra moneda un montón de veces y vemos *h* caras y *t* cruces. El teorema de Bayes (y otros cálculos matemáticos demasiado aburridos como para explicarlos aquí) nos dice que la distribución posterior para *p* es de nuevo una distribución beta, pero con los parámetros `alpha + h` y `beta + t`.

Nota: No es una coincidencia que la distribución posterior sea de nuevo una distribución beta. El número de caras viene dado por una distribución binomial,

y la beta es la previa conjugada[4] a la distribución binomial. Esto significa que, siempre que actualicemos una beta previa utilizando observaciones de la correspondiente binomial, obtendremos una posterior beta.

Digamos que lanzamos la moneda 10 veces y solo vemos 3 caras. Si empezáramos con la anterior uniforme (negándonos en cierto sentido a tomar partido sobre que la moneda sea justa o no), nuestra distribución posterior sería una Beta(4, 8), centrada en torno a 0,33. Como consideramos todas las probabilidades igualmente probables, nuestro mejor intento es próximo a la probabilidad observada.

Si empezáramos con una Beta(20, 20) (expresando la creencia de que la moneda era más o menos justa), nuestra distribución posterior sería una Beta(23, 27), centrada en torno a 0,46, indicando una creencia revisada de que quizá la moneda tiene una ligera tendencia hacia cruz.

Pero, si empezáramos con una Beta(30, 10) (expresando la creencia de que la moneda tiende a sacar cara un 75 % de las veces), nuestra distribución posterior sería una Beta(33, 17), centrada en torno a 0,66. En ese caso, seguiríamos creyendo en un desequilibrio hacia cara, pero con menos vehemencia que al principio. Estas tres diferentes posteriores se trazan en la figura 7.2.

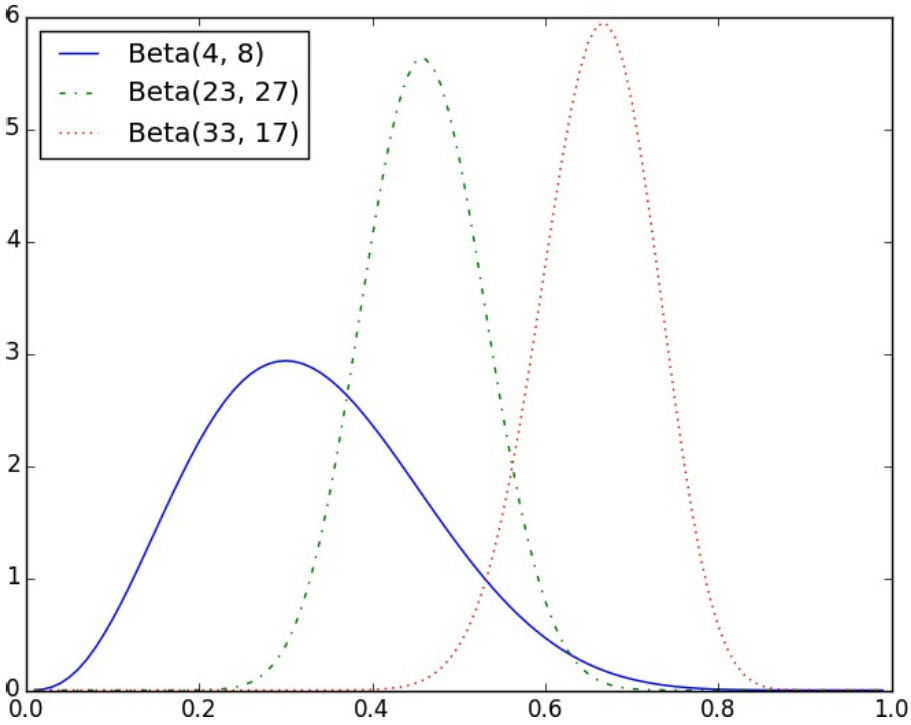


Figura 7.2. Posteriores surgiendo de distintas previas.

Si tirásemos la moneda más y más veces, la previa importaría cada vez menos hasta que al final tendríamos (casi) la misma distribución posterior sin importar la anterior con la que hubiéramos empezado.

Por ejemplo, sin importar la tendencia que nos parecía que pudiera tener la moneda en un principio, sería difícil mantener esa creencia tras ver 1.000 caras después de 2.000 lanzamientos, a menos que fuéramos unos lunáticos que eligiéramos algo parecido a una previa Beta(1000000,1).

Lo interesante es que esto nos permite realizar afirmaciones de probabilidad sobre las hipótesis: “Basándonos en la previa y en los datos observados, solo hay un 5 % de posibilidades de que la probabilidad de que la moneda saque cara esté entre el 49 % y el 51 %”. Esto es filosóficamente muy distinto a una aseveración como “si la moneda fuera justa, esperaríamos observar datos tan extremos solo el 5 % de las veces”.

Utilizar la inferencia bayesiana para probar hipótesis se considera un poco controvertido (en parte porque las matemáticas pueden llegar a ser bastante complicadas, y en parte debido a la naturaleza subjetiva de elegir una previa). No la utilizaremos más en este libro, pero es bueno conocerla.

Para saber más

- Apenas hemos arañado la superficie de lo que debería saber sobre inferencia estadística. Los libros recomendados al final del capítulo 5 entran mucho más en detalle al respecto.
- Coursera ofrece un curso de análisis de datos e inferencia estadística, en https://www.coursera.org/specializations/statistics, que trata muchos de estos temas.

---

[1] https://es.wikipedia.org/wiki/Correcci%C3%B3n_por_continuidad.

[2] https://www.nature.com/news/scientific-method-statistical-errors-1.14700.

[3] https://www.iro.umontreal.ca/~dift3913/cours/papers/cohen1994_The_earth_is_round

[4] https://www.johndcook.com/blog/conjugate_prior_diagram/.

8 Descenso de gradiente

Los que presumen de su ascendencia se jactan de lo que deben a los demás.

—Séneca

Cuando estemos haciendo ciencia de datos, muchas veces intentaremos encontrar el mejor modelo para una determinada situación. Generalmente "mejor" quiere decir algo así como "minimiza el error de sus predicciones" o "maximiza la probabilidad de los datos". En otras palabras, representará la solución a algún tipo de problema de optimización.

Esto significa que tendremos que resolver unos cuantos problemas de optimización; en particular, tendremos que hacerlo desde el principio. Nuestro enfoque para ello será una técnica denominada descenso de gradiente, que se presta a la perfección a un tratamiento iniciado desde cero. Quizá no resulte superinteresante en sí mismo, pero sí nos permitirá hacer cosas apasionantes a lo largo del libro, así que les pido un poco de paciencia.

La idea tras el descenso de gradiente

Supongamos que tenemos una función `f` cuya entrada es un vector de números reales y cuya salida es un solo número real. Una función como esta sencilla es algo así:

```
from scratch.linear_algebra import Vector, dot
def sum_of_squares(v: Vector) -> float:
   """Computes the sum of squared elements in v"""
   return dot(v, v)
```

En muchas ocasiones, tendremos que maximizar o minimizar este tipo de funciones. Es decir, tendremos que encontrar la entrada `v` que produzca el mayor (o menor) valor posible.

Para funciones como la nuestra, el gradiente (si se acuerda del cálculo que estudió en su día, es el vector de las derivadas parciales) proporciona la dirección de entrada en la que la función aumenta a mayor velocidad (si no se acuerda, créase lo que le digo, o busque en Internet).

Según esto, un método para maximizar una función es elegir un punto de inicio aleatorio, calcular el gradiente, dar un pequeño paso en la dirección del gradiente (es decir, la dirección que hace que la función aumente al máximo) y repetir con el nuevo punto de inicio. De forma similar, se puede minimizar una función dando pequeños pasos en la dirección opuesta, como muestra la figura 8.1.

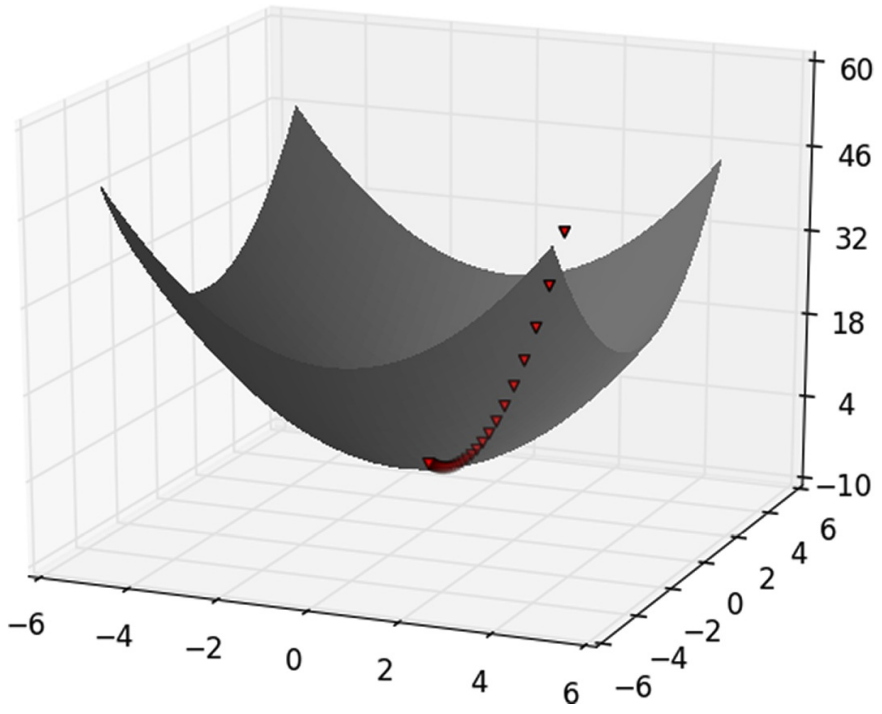


Figura 8.1. Hallar un mínimo utilizando el descenso de gradiente.

Nota: Si una función tiene un mínimo global único, es probable que este procedimiento lo encuentre. Si tiene varios mínimos (locales), quizá lo que el procedimiento "encuentre" sea el mínimo erróneo de todos ellos, en cuyo caso se podría volver a ejecutar desde distintos puntos de inicio. Si una función no tiene mínimo, entonces es posible que el procedimiento siga ejecutándose para siempre.

Estimar el gradiente

Si `f` es una función de una sola variable, su derivada en un punto `x` mide cómo cambia `f(x)` cuando le aplicamos un pequeño cambio a `x`. La derivada se define como el límite de los cocientes de diferencias:

```
from typing import Callable
def difference_quotient(f: Callable[[float], float],
                        x: float,
                        h: float) -> float:
   return (f(x + h)–f(x)) / h
```

Cuando `h` se aproxima a cero.

(Muchos aspirantes a estudiantes de cálculo se han visto obstaculizados por la definición matemática de límite, que es preciosa, pero puede resultar ciertamente intimidatoria. Aquí haremos trampas y simplemente diremos que "límite" significa lo que todos piensan que significa).

La derivada es la pendiente de la tangente en (x, $f(x)$), mientras que el cociente de diferencias es la pendiente de la línea no tan tangente que cruza ($x + h$, $f(x + h)$). A medida que h es más pequeño, la línea no tan tangente se acerca cada vez más a la línea tangente (figura 8.2).

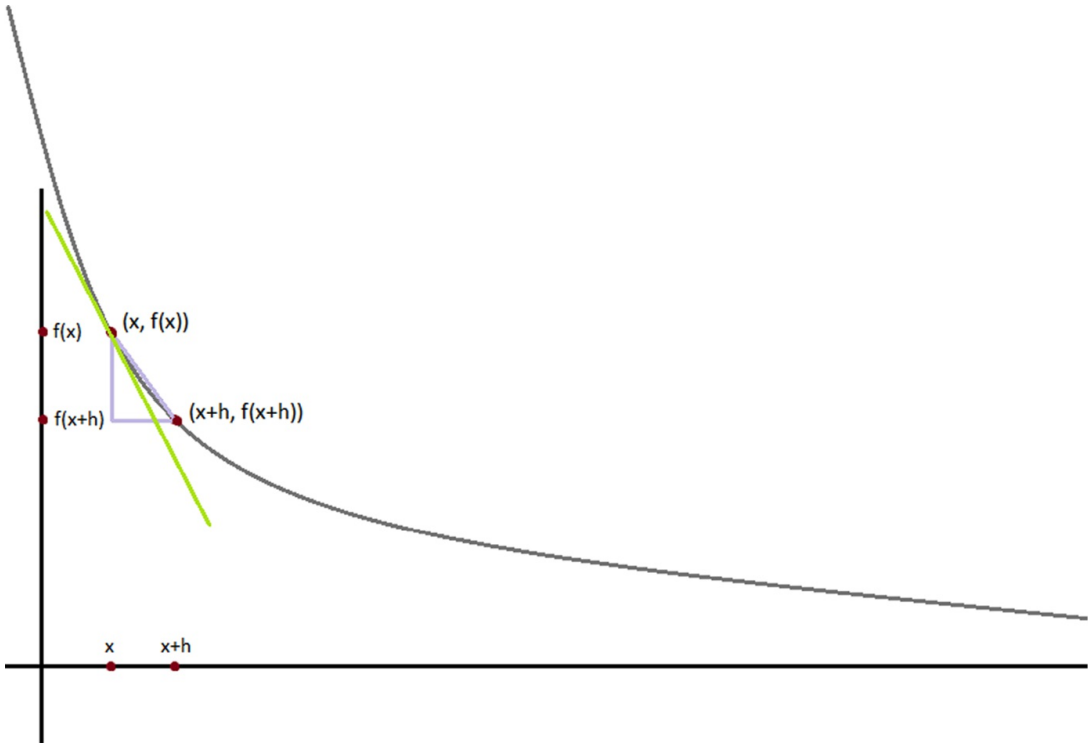


Figura 8.2. Aproximar una derivada con un cociente de diferencias.

Para muchas funciones es fácil calcular las derivadas con exactitud. Por ejemplo, la función `square`:

```
def square(x: float) -> float:
   return x * x
```

Tiene la derivada:

```
def derivative(x: float) -> float:
   return 2 * x
```

Que nos resulta sencillo comprobar calculando de manera explícita el cociente de diferencias y tomando el límite (lo que no requiere nada más que un poco de álgebra de instituto).

¿Qué pasa si no podemos (o no queremos) encontrar el gradiente? Aunque no podemos tomar límites en Python, podemos estimar derivadas evaluando el cociente de diferencias para un valor e muy pequeño. La figura 8.3 muestra los resultados de una estimación así:

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for x in xs]
# se traza para mostrar que son básicamente lo mismo
import matplotlib.pyplot as plt
plt.title("Actual Derivatives vs. Estimates")
plt.plot(xs, actuals, ‘rx’, label=’Actual’)             # rojo x
plt.plot(xs, estimates, ‘b+’, label=’Estimate’)         # azul +
plt.legend(loc=9)
plt.show()
```

Cuando `f` es una función de muchas variables, tiene varias derivadas parciales, cada una de las cuales indica cómo cambia `f` cuando realizamos pequeñas modificaciones en tan solo una de las variables de entrada.

Calculamos su *i* derivada parcial tratándola como una función solo de su *i* variable, manteniendo fijas el resto de variables:

```
def partial_difference_quotient(f: Callable[[Vector], float],
                      v: Vector,
                      i: int,
                      h: float) -> float:
  """Returns the i-th partial difference quotient of f at v"""
  w = [v_j + (h if j == i else 0)     # suma h al i elemento de v
     for j, v_j in enumerate(v)]
  return (f(w)–f(v)) / h
```

Tras lo cual podemos estimar el gradiente del mismo modo:

```
def estimate_gradient(f: Callable[[Vector], float],
     v: Vector,
     h: float = 0.0001):
  return [partial_difference_quotient(f, v, i, h)
       for i in range(len(v))]
```

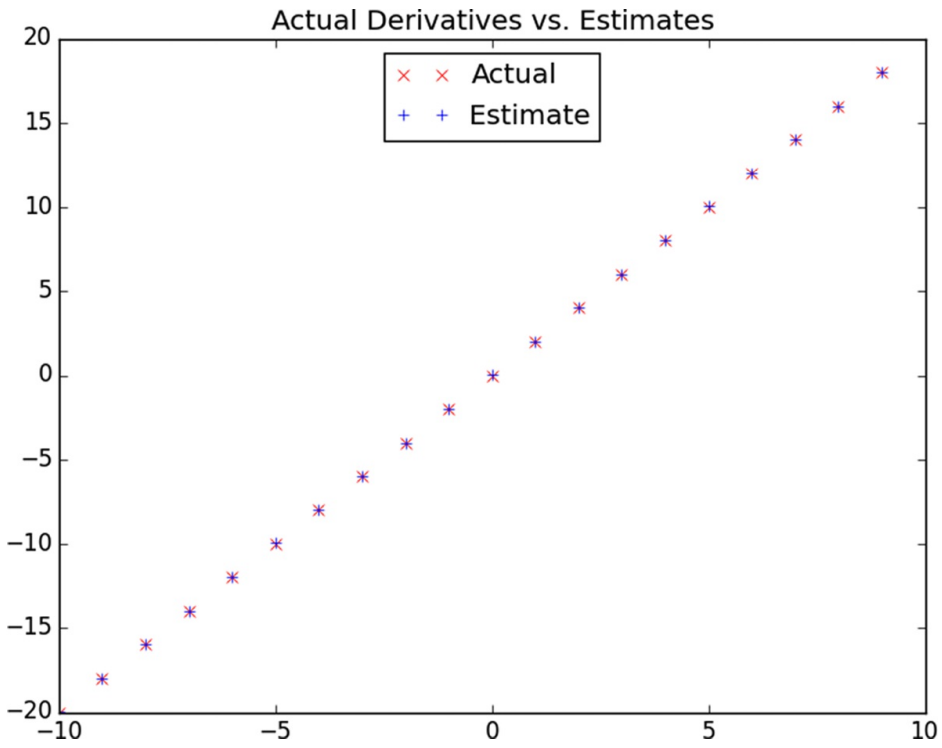


Figura 8.3. La bondad de la aproximación del cociente de diferencias.

Nota: Un inconveniente a tener en cuenta de este método de "estimación mediante cocientes de diferencias" es que resulta caro desde el punto de vista computacional. Si `v` tiene una longitud *n*, `estimate_gradient` tiene que evaluar `f` en 2*n* entradas distintas. Si estamos estimando gradientes repetidamente, estaremos trabajando demasiado. En todo lo que hagamos, emplearemos las matemáticas para calcular nuestras funciones de gradiente de manera explícita.

Utilizar el gradiente

Es fácil darse cuenta de que la función `sum_of_squares` es menor cuando su entrada `v` es un vector de ceros. Pero imaginemos que no sabíamos eso. Utilicemos los gradientes para hallar el mínimo entre todos los vectores tridimensionales. Elegiremos simplemente un punto de inicio aleatorio y después daremos pequeños pasos en la dirección opuesta al gradiente hasta alcanzar un punto en el que el gradiente sea muy pequeño:

```
import random
from scratch.linear_algebra import distance, add, scalar_multiply
def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
   """Moves 'step_size' in the 'gradient' direction from 'v'"""
   assert len(v) == len(gradient)
   step = scalar_multiply(step_size, gradient)
   return add(v, step)
def sum_of_squares_gradient(v: Vector) -> Vector:
   return [2 * v_i for v_i in v]
# elige un punto de inicio aleatorio
v = [random.uniform(-10, 10) for i in range(3)]
for epoch in range(1000):
   grad = sum_of_squares_gradient(v)       # calcula el gradiente en v
   v = gradient_step(v, grad, -0.01)       # da un paso de gradiente negativo
   print(epoch, v)
assert distance(v, [0, 0, 0]) < 0.001      # v debería estar cerca de 0
```

Si ejecutamos esto, veremos que siempre termina con un valor `v` muy próximo a `[0,0,0]`. Cuántas más veces se ejecute, más se acercará.

Elegir el tamaño de paso adecuado

Aunque los motivos para alejarnos del gradiente están claros, lo que no queda claro es lo lejos que queremos llegar. Sin duda, elegir el tamaño de paso correcto es más un arte que una ciencia. Entre las opciones más conocidas están las siguientes:

- Utilizar un tamaño de paso fijo.
- Ir encogiendo el tamaño de paso gradualmente con el tiempo.
- En cada paso, elegir el tamaño de paso que minimice el valor de la función objetivo.

El último método suena muy bien, pero, en la práctica, es un cálculo muy costoso. Para simplificar las cosas, utilizaremos la mayoría de las veces un tamaño de paso fijo. El tamaño de paso que “funcione” depende del problema: demasiado pequeño, y el descenso de gradiente se mantendrá para siempre; demasiado grande, y tendremos que dar pasos enormes que podrían

lograr que la función que nos ocupa sea cada vez más grande o incluso que se quede sin definir. Por lo tanto, hay que experimentar.

Utilizar descenso de gradiente para ajustar modelos

En este libro, utilizaremos el descenso de gradiente para ajustar modelos paramétricos a los datos. Lo más habitual es que tengamos un conjunto de datos y un modelo (hipotético) para los datos que depende (de una forma diferenciada) de uno o más parámetros. También tendremos una función de pérdida que mide lo bien que se ajusta el modelo a nuestros datos (menor es mejor). Si pensamos que nuestros datos son fijos, entonces nuestra función de pérdida nos indica lo buenos o malos que son los parámetros de un modelo cualquiera. Esto significa que podemos utilizar el descenso de gradiente para encontrar los parámetros del modelo que minimicen al máximo la pérdida. Veamos un sencillo ejemplo:

```
# x va de -50 a 49, y es siempre 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
```

En este caso, conocemos los parámetros de la relación lineal entre `x` e `y`, pero imaginemos que queremos descubrirlos a partir de los datos. Utilizaremos el descenso de gradiente para hallar la pendiente y la intersección que minimizan el error cuadrático medio.

Empezaremos con una función que determina el gradiente basándose en el error a partir de un solo punto de datos:

```
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
   slope, intercept = theta
   predicted = slope * x + intercept     # La predicción del modelo.
   error = (predicted-y)                 # el error es (previsto-real).
   squared_error = error ** 2            # Minimizaremos el error cuadrático
   grad = [2 * error * x, 2 * error]     # usando su gradiente.
   return grad
```

Pensemos en lo que significa el gradiente. Imaginemos que para un cierto

`x` nuestra predicción es demasiado grande. En ese caso, el `error` es positivo. El segundo término de gradiente, `2 * error`, es positivo, lo que refleja el hecho de que pequeños incrementos harán que la predicción (ya demasiado grande) sea aún más grande, lo que provocará que el error cuadrático (para este `x`) sea aún mayor.

El primer término de gradiente, `2 * error * x`, tiene el mismo signo que `x`. Por supuesto que, si `x` es positivo, los pequeños incrementos en la pendiente harán de nuevo que la predicción (y de ahí el error) sea más grande. Si `x` es negativo, sin embargo, esos pequeños incrementos harán que la predicción (y por lo tanto el error) sea más pequeña.

Pero ese cálculo está hecho para un solo punto de datos. Para el conjunto completo miraremos el error cuadrático medio; el gradiente del error cuadrático medio no es más que la media de los gradientes individuales.

Así, esto es lo que vamos a hacer:

1. Empezar con un valor aleatorio para `theta`.
2. Calcular la media de los gradientes.
3. Ajustar `theta` en esa dirección.
4. Repetir.

Tras muchos *epochs* (como llamamos a cada pasada por el conjunto de datos), descubriríamos algo parecido a los parámetros correctos:

```
from scratch.linear_algebra import vector_mean
# Empieza con valores aleatorios para pendiente e intersección
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
learning_rate = 0.001
for epoch in range(5000):
   # Calcula la media de los gradientes
   grad = vector_mean([linear_gradient(x, y, theta) for x, y in inputs])
   # Da un paso en esa dirección
   theta = gradient_step(theta, grad, -learning_rate)
   print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Descenso de gradiente en minilotes y estocástico

Un inconveniente del método anterior es que hemos tenido que evaluar los gradientes en el conjunto de datos entero antes de poder dar un paso de gradiente y actualizar nuestros parámetros. En este caso estaba bien, porque nuestro conjunto de datos contenía solamente 100 pares y el cálculo del gradiente resultó barato.

Pero otros modelos tendrán con frecuencia grandes conjuntos de datos y caros cálculos de gradientes. En ese caso, nos interesará dar pasos de gradiente más a menudo.

Podemos hacerlo utilizando una técnica denominada descenso de gradiente en minilotes (o *minibatch*), en la que calculamos el gradiente (y damos un paso de gradiente) basándonos en un "minilote" muestreado del conjunto de datos principal:

```
from typing import TypeVar, List, Iterator
T = TypeVar('T')                          # nos permite escribir funciones
                                          "genéricas"
def minibatches(dataset: List[T],
                batch_size: int,
                shuffle: bool = True) -> Iterator[List[T]]:
  """Generates 'batch_size'-sized minibatches from the dataset"""
  # inicia índices 0, batch_size, 2 * batch_size, ...
  batch_starts = [start for start in range(0, len(dataset), batch_size)]
  if shuffle:                             # mezcla los lotes
  random.shuffle(batch_starts)
  for start in batch_starts:
    end = start + batch_size
    yield dataset[start:end]
```

Nota: `TypeVar (T)` nos permite crear una función "genérica", que dice que nuestro conjunto de datos puede ser una lista de cualquier tipo sencillo (`str`, `int`, `list`, lo que sea), pero, sea cual sea el tipo, los resultados serán lotes de él.

Podemos resolver nuestro problema de nuevo utilizando minilotes:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
```

```
for epoch in range(1000):
   for batch in minibatches(inputs, batch_size=20):
      grad = vector_mean([linear_gradient(x, y, theta) for x, y in batch])
      theta = gradient_step(theta, grad, -learning_rate)
   print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Otra variante es el descenso de gradiente estocástico, en el que se dan pasos de gradiente basados en un ejemplo de entrenamiento cada vez:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(100):
   for x, y in inputs:
      grad = linear_gradient(x, y, theta)
      theta = gradient_step(theta, grad, -learning_rate)
   print(epoch, theta)
slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, “intercept should be about 5”
```

En este problema, el descenso de gradiente estocástico encuentra los parámetros óptimos en un número de *epochs* mucho menor. Pero siempre hay inconvenientes. Basar los pasos de gradiente en pequeños minilotes (o en puntos de datos sencillos) permite dar más pasos, pero el gradiente para un solo punto podría residir en una dirección muy distinta a la del gradiente para el conjunto completo de datos.

Además, si no estuviéramos creando nuestra álgebra lineal desde cero, se producirían mejoras en el rendimiento por “vectorizar” nuestros cálculos a lo largo de lotes en lugar de calcular el gradiente un punto cada vez.

A lo largo del libro, jugaremos a buscar y encontrar tamaños de lote y de paso óptimos.

Nota: La terminología para los distintos tipos de descensos de gradiente no es uniforme. El método “calcular el gradiente para el conjunto de datos completo” se suele denominar descenso de gradiente en lotes, y algunas personas dicen descenso de gradiente estocástico cuando se quieren referir a la versión de minilotes (cuya versión “un punto cada vez” es un caso especial).

Para saber más

- ¡Siga leyendo! Utilizaremos el descenso de gradiente para resolver problemas en el resto del libro.
- A estas alturas, es casi seguro que ya se ha hartado de leerme recomendándole que lea libros de texto. Si le sirve de consuelo, *Active Calculus 1.0*, en `https://scholarworks.gvsu.edu/books/10/`, de Matthew Boelkins, David Austin y Steven Schlicker (Bibliotecas de la Universidad Estatal de Grand Valley), parece más interesante que los libros de texto con los que yo aprendí.
- Sebastian Ruder tiene una entrada épica en su blog, en `http://ruder.io/optimizing-gradient-descent/index.html`, comparando el descenso de gradiente y sus distintas variantes.

9 Obtener datos

Para escribirlo, necesité tres meses; para concebirlo, tres minutos; para recoger los datos contenidos en él, toda mi vida.

—F. Scott Fitzgerald

Para ser científico de datos se necesitan datos. De hecho, como científico de datos se pasará una fracción indecorosamente grande de su tiempo adquiriendo, limpiando y transformando datos. Si no hay más remedio, puede escribirlos usted mismo (o si tiene minions a su disposición, mejor que lo hagan ellos), pero este no es un buen uso de su tiempo. En este capítulo, veremos distintas formas de obtener datos en Python y en los formatos adecuados.

stdin y stdout

Si ejecutamos nuestros *scripts* de Python en la línea de comandos, se pueden canalizar los datos a través de ellos utilizando `sys.stdin` y `sys.stdout`. Por ejemplo, esta es una secuencia de comandos que lee líneas de texto y devuelve las que coinciden con una expresión regular:

```
# egrep.py
import sys, re
# sys.argv es la lista de argumentos de línea de comandos
# sys.argv[0] es el nombre del propio programa
# sys.argv[1] será el regex especificado en la línea de comandos
regex = sys.argv[1]
# por cada línea pasada al script
for line in sys.stdin:
   # si coincide con el regex, lo graba en stdout
   if re.search(regex, line):
      sys.stdout.write(line)
```

Y aquí tenemos un fragmento de código que cuenta las líneas que recibe y devuelve el total:

```
# line_count.py
import sys
count = 0
for line in sys.stdin:
   count += 1
# print va a sys.stdout
print(count)
```

Podríamos entonces utilizar estos *scripts* para contar cuántas líneas de un archivo contienen números. En Windows emplearíamos:

```
type SomeFile.txt | python egrep.py “[0-9]” | python line_count.py
```

Mientras que en un sistema Unix se transformaría en:

```
cat SomeFile.txt | python egrep.py “[0-9]” | python line_count.py
```

El carácter | (barra vertical) es el denominado *pipe*, que significa “utilizar la salida del comando izquierdo como entrada del comando derecho”. De este modo se pueden construir *pipelines* (o tuberías) de proceso de datos.

Nota: Si utilizamos Windows, es probable que podamos omitir la parte de Python de este comando:

```
type SomeFile.txt | egrep.py “[0-9]” | line_count.py
```

Si trabajamos en un sistema Unix, hacer esto mismo requiere un par de pasos adicionales.[1] Primero, añadimos un “shebang” como primera línea del *script* `#!/usr/bin/env python`. Después, en la línea de comandos, utilizamos `chmod` x egrep.py++ para convertir el archivo en ejecutable.

De forma similar, este es un *script* que cuenta las palabras de su entrada y devuelve las más comunes:

```
# most_common_words.py
import sys
from collections import Counter
# pasa el número de palabras como primer argumento
```

```
try:
   num_words = int(sys.argv[1])
except:
   print("usage: most_common_words.py num_words")
   sys.exit(1)                              # un código de salida no cero indica
                                            error
counter = Counter(word.lower()                  # palabras en minúscula
             for line in sys.stdin
             for word in                        # divide por espacios
             line.strip().split()
             if word)                           # salta las 'palabras' vacías
for word, count in counter.most_common(num_words):
   sys.stdout.write(str(count))
   sys.stdout.write("\t")
   sys.stdout.write(word)
   sys.stdout.write("\n")
```

Después de esto, se podría hacer algo así:

```
$ cat the_bible.txt | python most_common_words.py 10
36397                    the
30031                    and
20163                    of
7154                     to
6484                     in
5856                     that
5421                     he
5226                     his
5060                     unto
4297                     shall
```

(Si utiliza Windows, emplee `type` en lugar de `cat`).

Nota: Si es un experto programador de Unix, probablemente estará familiarizado con una gran variedad de herramientas de línea de comandos (por ejemplo, `egrep`), integradas en el sistema operativo, que es preferible crear desde cero. Aun así, es bueno saber que puede si lo necesita.

Leer archivos

También es posible leer archivos y escribir en ellos de forma explícita directamente en el código. Python simplifica bastante el trabajo con archivos.

Conocimientos básicos de los archivos de texto

El primer paso para trabajar con un archivo de texto es obtener un objeto archivo utilizando `open`:

```
# ‘r’ significa solo lectura, se da por sentado si se omite
file_for_reading = open(‘reading_file.txt’, ‘r’)
file_for_reading2 = open(‘reading_file.txt’)
# ‘w’ es escribir — ¡destruirá el archivo si ya existe!
file_for_writing = open(‘writing_file.txt’, ‘w’)
# ‘a’ es añadir – para añadir al final del archivo
file_for_appending = open(‘appending_file.txt’, ‘a’)
# no olvide cerrar sus archivos al terminar
file_for_writing.close()
```

Como es fácil olvidarse de cerrar los archivos, siempre se deben utilizar con un bloque `with`, al final del cual se cerrarán automáticamente:

```
with open(filename) as f:
   data = function_that_gets_data_from(f)
# en este momento f ya se ha cerrado, así que no trate de usarlo
process(data)
```

Si hace falta leer un archivo de texto completo, basta simplemente con pasar varias veces por las líneas del archivo utilizando `for`:

```
starts_with_hash = 0
with open(‘input.txt’) as f:
   for line in f:                  # mira cada línea del archivo
      if re.match("^#",line):      # usa un regex para ver si empieza por ‘#’
         starts_with_hash += 1     # si es así, suma 1 al total
```

Todas las líneas obtenidas así terminan en un carácter de línea nueva, así que con frecuencia nos interesará limpiarlas con un `strip` antes de hacer nada con ellas.

Por ejemplo, imaginemos que tenemos un archivo lleno de direcciones de correo electrónico, una por línea, y necesitamos generar un histograma de los dominios. Las reglas para extraer dominios correctamente son algo sutiles (vea, por ejemplo, la lista de sufijos públicos `https://publicsuffix.org`), pero una buena primera aproximación es coger las partes de las direcciones de correo que vienen después del @ (lo que da la respuesta equivocada con direcciones como joel@mail.datasciencester.com, pero solo para este ejemplo estamos dispuestos a vivir con ello):

```
def get_domain(email_address: str) -> str:
   """Split on '@' and return the last piece"""
   return email_address.lower().split("@")[-1]

# un par de pruebas
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencester.com') == 'm.datasciencester.com'

from collections import Counter

with open('email_addresses.txt', 'r') as f:
   domain_counts = Counter(get_domain(line.strip())
                   for line in f
                   if "@" in line)
```

Archivos delimitados

Las direcciones de correo electrónico hipotéticas que acabamos de procesar solo tenían una dirección por línea. Lo más habitual es que los archivos tengan muchos datos en cada línea. Estos archivos suelen estar separados (o delimitados) por comas o tabuladores: cada línea tiene varios campos, con una coma o un tabulador indicando el lugar en el que termina un campo y empieza el siguiente.

Esto empieza a complicarse cuando se tienen campos que incluyen comas, tabuladores y líneas nuevas (algo que ocurrirá forzosamente). Por esta razón, no conviene nunca intentar analizarlos uno mismo. Es mejor utilizar el módulo `csv` de Python (o la librería pandas, o cualquier otra diseñada para leer archivos delimitados por comas o tabuladores).

Advertencia: ¡Nunca analice usted solo un archivo delimitado por comas! ¡Se cargará los casos límite!

Si el archivo no tiene encabezados (lo que significa que probablemente nos interese cada fila como una `list`, y lo que además nos obliga a saber lo que hay en cada columna), se puede utilizar `csv.reader` para pasar varias veces por las filas, cada una de las cuales será una lista adecuadamente dividida.

Si tuviéramos por ejemplo un archivo delimitado por tabuladores de precios de acciones:

```
6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB  64.5
6/19/2014 AAPL 91.86
6/19/2014 MSFT 41.51
6/19/2014 FB  64.34
```

Podríamos procesar las líneas con:

```
import csv
with open('tab_delimited_stock_prices.txt') as f:
   tab_reader = csv.reader(f, delimiter='\t')
   for row in tab_reader:
      date = row[0]
      symbol = row[1]
      closing_price = float(row[2])
      process(date, symbol, closing_price)
```

Si el archivo tiene encabezados:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

Se puede omitir la fila del encabezado con una llamada inicial a `reader.next`, o bien obtener cada fila como un `dict` (con los encabezados como claves) utilizando `csv.DictReader`:

```
with open(‘colon_delimited_stock_prices.txt’) as f:
   colon_reader = csv.DictReader(f, delimiter=’:’)
   for dict_row in colon_reader:
      date = dict_row["date"]
      symbol = dict_row["symbol"]
      closing_price = float(dict_row["closing_price"])
      process(date, symbol, closing_price)
```

Aunque el archivo no tuviera encabezados, aún podríamos utilizar `DictReader` pasándole las claves como un parámetro `fieldnames`.

También se pueden escribir datos delimitados utilizando `csv.writer`:

```
todays_prices = {‘AAPL’: 90.91, ‘MSFT’: 41.68, ‘FB’: 64.5 }
with open(‘comma_delimited_stock_prices.txt’, ‘w’) as f:
   csv_writer = csv.writer(f, delimiter=’,’)
   for stock, price in todays_prices.items():
        csv_writer.writerow([stock, price])
```

`csv.writer` hará lo correcto si los campos contienen comas. Escribirlos a mano probablemente no servirá. Por ejemplo, si intentamos lo siguiente:

```
results = [["test1", "success", "Monday"],
        ["test2", "success, kind of", "Tuesday"],
        ["test3", "failure, kind of", "Wednesday"],
        ["test4", "failure, utter", "Thursday"]]
# ¡no haga esto!
with open(‘bad_csv.txt’, ‘w’) as f:
   for row in results:
      f.write(",".join(map(str,            # ¡podría contener demasiadas comas!
      row)))
      f.write("\n")                        # ¡la fila podría tener líneas
                                           nuevas!
```

Terminará con un archivo `.csv` parecido a esto:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

Al que nadie podrá dar sentido.

Raspado web

Otra forma de conseguir datos es extrayéndolos de páginas web mediante el método del raspado web (*web scraping*). Parece que conseguir páginas web es bastante fácil; otra cosa es obtener de ellas información estructurada con sentido.

HTML y su análisis

Las páginas de la web están escritas en HTML, donde el texto (preferiblemente) se marca con elementos y sus atributos:

```
<html>
  <head>
    <title>Una página web</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Ciencia de datos</p>
  </body>
</html>
```

En un mundo perfecto, en el que todas las páginas web estuvieran marcadas semánticamente para nuestro beneficio, podríamos extraer datos utilizando reglas como "encuentra el elemento `<p>` cuyo `id` es `subject` y devuelve el texto que contiene". Pero, en la realidad, HTML no suele estar bien formado, y no digamos bien comentado. Esto significa que necesitaremos ayuda para darle sentido.

Para extraer datos de HTML, utilizaremos la librería Beautiful Soup,[2] que construye un árbol con los distintos elementos de una página web y ofrece una sencilla interfaz para acceder a ellos. La última versión en el momento de escribir este libro es Beautiful Soup 4.6.0, que es la que utilizaremos. También vamos a emplear la librería Requests,[3] una forma mucho más interesante de hacer peticiones HTTP que nada que esté integrado en Python.

El analizador de HTML integrado en Python no es tan tolerante, es decir, que no siempre se lleva bien con código HTML que no esté perfectamente

formado. Por esa razón, instalaremos además el analizador `html5lib`.

Asegurándonos de estar en el entorno virtual correcto, instalamos las librerías:

```
python -m pip install beautifulsoup4 requests html5lib
```

Para utilizar Beautiful Soup, pasamos una cadena de texto que contiene HTML a la función `BeautifulSoup`. En nuestros ejemplos, este será el resultado de una llamada a `requests.get`:

```
from bs4 import BeautifulSoup
import requests
# Pongo el archivo HTML en GitHub. Para encajar
# la URL en el libro tuve que dividirla en dos líneas.
# Recuerde que las cadenas de texto whitespace-separated se concatenan.
url = ("https://raw.githubusercontent.com/"
     "joelgrus/data/master/getting-data.html")
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

Tras de lo cual podemos llegar bastante lejos utilizando unos cuantos métodos sencillos.

Normalmente trabajaremos con objetos `Tag`, que corresponden a las etiquetas que representan la estructura de una página HTML.

Por ejemplo, para encontrar la primera etiqueta `<p>` (y su contenido), se puede utilizar:

```
first_paragraph = soup.find('p')      # o simplemente soup.p
```

Se puede obtener el contenido de texto de una `Tag` utilizando su propiedad `text`:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

Y se pueden extraer los atributos de una etiqueta tratándola como un `dict`:

```
first_paragraph_id = soup.p['id']           # da un KeyError si no hay 'id'
first_paragraph_id2 = soup.p.get('id')      # devuelve None si no hay 'id'
```

Se pueden conseguir varias etiquetas al mismo tiempo del siguiente modo:

```
all_paragraphs = soup.find_all('p')      # o simplemente soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Con frecuencia nos vendrá bien encontrar etiquetas con una determinada `class`:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                  if 'important' in p.get('class', [])]
```

Y también es posible combinar estos métodos para implementar una lógica más elaborada. Por ejemplo, si queremos encontrar todos los elementos `<span>` contenidos dentro de un elemento `<div>`, podríamos hacer lo siguiente:

```
# Atención: devolverá el mismo <span> varias veces
# si está dentro de varios <div>s.
# Sea más listo si ocurre esto.
spans_inside_divs = [span
               for div in soup('div')        # por cada <div> de la página
               for span in div('span')]      # halla los <span> contenidos
```

Solo este montón de funciones ya nos permitirá hacer bastante. Si finalmente hay que hacer cosas más complicadas (o simplemente si tiene curiosidad), eche un vistazo a la documentación que encontrará en https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

Sin duda alguna, los datos importantes no van a estar etiquetados, claro está, como `class="important"`. Será necesario inspeccionar con detalle el código fuente HTML, razonar sobre la lógica de selección y preocuparse de los casos límite para estar seguros de que los datos son correctos. Veamos un ejemplo.

Ejemplo: Controlar el congreso

El vicepresidente de Política de DataSciencester está preocupado por las posibles regulaciones del sector de la ciencia de datos y le pide que cuantifique lo que dice el Congreso de los Estados Unidos sobre el tema. En especial quiere que encuentre a todos los representantes que tengan notas de prensa sobre “datos”.

En el momento de la publicación de este libro, existe la página `https://www.house.gov/representatives` con enlaces a los sitios web de todos los representantes.

Si visualizamos el código, todos los enlaces tienen este aspecto:

```
<td>
   <a href="https://jayapal.house.gov">Jayapal, Pramila</a>
</td>
```

Empecemos recopilando todas las URL a las que se enlaza desde esa página:

```
from bs4 import BeautifulSoup
import requests
url = "https://www.house.gov/representatives"
text = requests.get(url).text
soup = BeautifulSoup(text, "html5lib")
all_urls = [a['href']
        for a in soup('a')
        if a.has_attr('href')]
print(len(all_urls))     # 965 para mí, demasiadas
```

Esto devuelve demasiadas URL. Si les echamos un vistazo, las que queremos empiezan con `http://` o `https://`, tienen después algún nombre y terminan con `.house.gov` o `.house.gov/`.

Es un buen momento para utilizar una expresión regular:

```
import re
# Debe empezar con http:// o https://
# Debe terminar con .house.gov o .house.gov/
regex = r"^https?://.*\.house\.gov/?$"
# ¡Escribamos algunas pruebas!
assert re.match(regex, "http://joel.house.gov")
assert re.match(regex, "https://joel.house.gov")
```

```
assert re.match(regex, "http://joel.house.gov/")
assert re.match(regex, "https://joel.house.gov/")
assert not re.match(regex, "joel.house.gov")
assert not re.match(regex, "http://joel.house.com")
assert not re.match(regex, "https://joel.house.gov/biography")
# Ahora aplicamos
good_urls = [url for url in all_urls if re.match(regex, url)]
print(len(good_urls))              # aun 862 para mi
```

Siguen siendo demasiados, ya que solamente hay 435 representantes. Si miramos la lista, hay muchos duplicados. Utilicemos `set` para deshacernos de ellos:

```
good_urls = list(set(good_urls))
print(len(good_urls))# solo 431 para mi
```

Siempre hay un par de escaños libres en la cámara, o quizá haya algún representante sin sitio web. En cualquier caso, con esto es suficiente. Al revisar los sitios, vemos que la mayoría de ellos tienen un enlace a notas de prensa. Por ejemplo:

```
html = requests.get('https://jayapal.house.gov').text
soup = BeautifulSoup(html, 'html5lib')
# Usa un conjunto porque los enlaces podrían aparecer varias veces.
links = {a['href'] for a in soup('a') if 'press releases' in a.text.lower()}
print(links)           # {'/media/press-releases'}
```

Hay que tener en cuenta que es un enlace relativo, es decir, tenemos que recordar el sitio del que se originó. Hagamos un poco de raspado:

```
from typing import Dict, Set
press_releases: Dict[str, Set[str]] = {}
for house_url in good_urls:
   html = requests.get(house_url).text
   soup = BeautifulSoup(html, 'html5lib')
   pr_links = {a['href'] for a in soup('a') if 'press releases'
                                 in a.text.lower()}
   print(f"{house_url}: {pr_links}")
   press_releases[house_url] = pr_links
```

Nota: Normalmente, no es muy correcto raspar un sitio libremente de esta forma. La mayoría de los sitios web tienen un archivo `robots.txt` que indica la frecuencia con la que se pueden extraer datos del sitio (y las rutas en las que se supone que no se debe hacer), pero, como es el Congreso, no hace falta que seamos especialmente educados.

Si los observamos según van apareciendo, veremos muchos `/media/press-releases` y `media-center/press-releases`, además de otras direcciones varias. Una de estas URL es `https://jayapal.house.gov/media/press-releases`.

Conviene recordar que nuestro objetivo es averiguar qué congresistas tienen notas de prensa que contienen "datos" (*data* en inglés). Escribiremos una función algo más general que verifique si una página de notas de prensa menciona un determinado término.

Visitando el sitio y revisando el código fuente, parece que haya un fragmento de cada nota de prensa dentro de una etiqueta `<p>`, de modo que utilizaremos esto como primer intento:

```
def paragraph_mentions(text: str, keyword: str) -> bool:
  """
  Returns True if a <p> inside the text mentions {keyword}
  """
  soup = BeautifulSoup(text, 'html5lib')
  paragraphs = [p.get_text() for p in soup('p')]
  return any(keyword.lower() in paragraph.lower()
         for paragraph in paragraphs)
```

Preparemos una prueba rápida de esto:

```
text = """<body><h1>Facebook</h1><p>Twitter</p>"""
assert paragraph_mentions(text, "twitter")     # está dentro de una <p>
assert not paragraph_mentions(text, "facebook")# no está dentro de una <p>
```

Finalmente, estamos listos para encontrar a los congresistas que buscábamos y dar sus nombres al vicepresidente:

```
for house_url, pr_links in press_releases.items():
  for pr_link in pr_links:
```

```
    url = f"{house_url}/{pr_link}"
    text = requests.get(url).text
    if paragraph_mentions(text, 'data'):
       print(f"{house_url}")
       break        # lista esta house_url
```

Al ejecutar este fragmento de código obtenemos una lista de 20 representantes. Probablemente otra persona obtenga un resultado distinto.

> **Nota:** Si revisamos las distintas páginas de "notas de prensa" (*press releases* en inglés), la mayoría están paginadas solo con 5 o 10 notas de prensa por página. Esto significa que solamente hemos recuperado las notas de prensa más recientes para cada congresista. Una solución más meticulosa habría pasado varias veces por las páginas y recuperado el texto completo de cada nota de prensa.

Utilizar API

Muchos sitios y servicios web ofrecen interfaces de programación de aplicaciones o API (*Application Programming Interfaces*), que permiten solicitar de manera explícita datos en un formato estructurado (¡lo que nos ahorra el problema de tener que rasparlos!).

JSON y XML

Como HTTP es un protocolo para transferir texto, los datos que se soliciten a través de una API web se tienen que serializar en un formato de cadena de texto. Con frecuencia esta serialización emplea la notación de objeto de JavaScript o JSON (*JavaScript Object Notation*). Los objetos de JavaScript son bastante parecidos a las clases `dict` de Python, lo que facilita la interpretación de sus representaciones de texto:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2019,
```

```
   "topics" : [ "data", "science", "data science"] }
```

Podemos analizar JSON utilizando el módulo `json` de Python. En particular, utilizaremos su función `loads`, que deserializa una cadena de texto que representa un objeto JSON y la transforma en un objeto Python:

```
import json
serialized = """{ "title" : "Data Science Book",
             "author" : "Joel Grus",
             "publicationYear" : 2019,
             "topics" : [ "data", "science", "data science"] }"""
# analiza JSON para crear un dict de Python
deserialized = json.loads(serialized)
assert deserialized["publicationYear"] == 2019
assert "data science" in deserialized["topics"]
```

A veces, un proveedor de API te odia y solamente ofrece respuestas en XML:

```
<Book>
   <Title>Data Science Book</Title>
   <Author>Joel Grus</Author>
   <PublicationYear>2014</PublicationYear>
   <Topics>
      <Topic>data</Topic>
      <Topic>science</Topic>
      <Topic>data science</Topic>
   </Topics>
</Book>
```

Puede utilizar Beautiful Soup para obtener datos de XML de forma parecida a como lo usamos para obtener datos de HTML; revise su documentación para más detalles.

Utilizar una API no autenticada

La mayoría de las API actuales exigen que primero se autentifique uno mismo antes de poder utilizarlas. Aunque no envidiamos esta política, crea una gran cantidad de información adicional que enturbia nuestra exposición.

Según esto, empezaremos echando un vistazo a la API de GitHub,[4] con la que podemos hacer algunas cosas sencillas sin necesitar de autenticación:

```
import requests, json
github_user = "joelgrus"
endpoint = f"https://api.github.com/users/{github_user}/repos"
repos = json.loads(requests.get(endpoint).text)
```

Debo decir que `repos` es una `list` de clases `dict` de Python, que representa cada una un repositorio público en mi cuenta de GitHub (coloque con toda libertad su nombre de usuario y obtenga su propio repositorio de GitHub; porque, tiene cuenta en GitHub, ¿verdad?).

Podemos utilizar esto para averiguar en qué meses y días del año es más probable que yo cree un repositorio. El único inconveniente es que las fechas de la respuesta son cadenas de texto:

```
"created_at": "2013-07-05T02:02:28Z"
```

Python no incluye un analizador de fechas demasiado bueno, así que tendremos que instalar uno:

```
python -m pip install python-dateutil
```

A partir del cual es probable que solo se necesite la función `dateutil.parser.parse`:

```
from collections import Counter
from dateutil.parser import parse
dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

De forma similar, se pueden conseguir los lenguajes de mis cinco últimos repositorios:

```
last_5_repositories = sorted(repos,
 key=lambda r: r["pushed_at"],
 reverse=True)[:5]
last_5_languages = [repo["language"]
                for repo in last_5_repositories]
```

Lo normal es que no trabajemos con API en este bajo nivel de "hacemos las solicitudes y analizamos las respuestas nosotros mismos". Uno de los beneficios de utilizar Python es que alguien ya ha creado una librería para prácticamente cualquier API a la que estemos interesados en acceder. Cuando lo han hecho bien, estas librerías pueden ahorrar buena parte del problema de averiguar los detalles más peliagudos del acceso API (pero, cuando no lo han hecho tan bien o cuando resulta que están basados en versiones difuntas de las correspondientes API, pueden provocar enormes dolores de cabeza).

No obstante, de vez en cuando habrá que desarrollar una librería de acceso API propia (o, lo más probable, depurar, porque la de otra persona no funcione), de modo que resulta positivo conocer parte de los detalles.

Encontrar API

Si nos hacen falta datos de un determinado sitio, hay que buscar en él una sección "desarrolladores", "*developers*" o "API" para obtener más información, y tratar de buscar en la web "python <nombredelsitio> api" para encontrar una librería.

Hay librerías para la API de Yelp, de Instagram, de Spotify, etc.

Si lo que queremos es una lista de API que tengan *wrappers* de Python, una que está muy bien es la de Real Python de GitHub.[5]

Y, si no se encuentra lo buscado, siempre quedará el raspado, el último refugio del científico de datos.

Ejemplo: Utilizar las API de Twitter

Twitter es una fuente fenomenal de datos para trabajar. Se puede utilizar para obtener noticias en tiempo real, para medir reacciones a acontecimientos actuales, para encontrar enlaces relacionados con determinados temas, etc. En definitiva, se puede utilizar prácticamente para cualquier cosa que se pueda imaginar, siempre y cuando se tenga acceso a sus datos, obviamente a través de sus API.

Para interactuar con las API de Twitter, vamos a emplear la librería

Twython[6] (`python -m pipinstall twython`). Existen bastantes librerías de Python para Twitter, pero con esta es con la que me ha ido mejor. ¡Le animo a que explore otras opciones!

Obtener credenciales

Para poder utilizar las API de Twitter, es necesario conseguir credenciales (para lo cual hay que tener una cuenta de Twitter, que probablemente tendrá si desea formar parte de la animada y amistosa comunidad #datascience de Twitter).

Advertencia: Como todas las instrucciones relacionadas con sitios web que no controlo, estas pueden quedar obsoletas en algún momento, pero espero que funcionen durante el tiempo suficiente (aunque ya han cambiado varias veces desde que empecé a escribir este libro, así que ¡buena suerte!).

Estos son los pasos a seguir:

1. Vaya a `https://developer.twitter.com/`.
2. Si no está ya dentro, haga clic en Sign in e introduzca su nombre de usuario de Twitter y su contraseña.
3. Haga clic en Apply para solicitar una cuenta de desarrollador.
4. Solicite acceso para uso personal.
5. Rellene la solicitud. Dispondrá de 300 palabras para explicar (en serio) por qué necesita el acceso, de modo que para pasarse del límite puede hablarles de este libro y de lo mucho que está disfrutando con su lectura.
6. Espere una cantidad de tiempo indefinida.
7. Si conoce a alguien que trabaje en Twitter, envíele un correo electrónico preguntándole si puede acelerar su solicitud. Si no, siga esperando.
8. En cuanto se la aprueben, vuelva a `https://developer.twitter.com/`, localice la sección Apps y haga clic en Create an app.
9. Rellene todos los campos necesarios (aquí también, si necesita texto

adicional para la descripción, puede hablar de este libro y de lo edificante que le está resultando).

10. Haga clic en CREATE.

Ahora su aplicación debería incluir una pestaña Keys and tokens con una sección Consumer API keys, donde se lista una "clave API" y una "clave secreta API". Tome nota de estas claves, porque las necesitará (y guárdelas bien; son como contraseñas).

> **Advertencia:** No comparta las claves, no las publique en su libro, ni las compruebe en su repositorio público de GitHub. Una solución sencilla es almacenarlas en un archivo `credenciales.json` que no se compruebe, y hacer que su código utilice `json.loads` para recuperarlas. Otra solución es almacenarlas en variables de entorno y utilizar `os.environ` para recuperarlas.

Utilizar Twython

La parte más difícil de utilizar la API de Twitter es autentificarse a uno mismo (de hecho, esto es lo más difícil en la mayoría de las API). Los proveedores de API quieren asegurarse de que está autorizado para acceder a sus datos y que no va a exceder sus límites de uso. También quieren saber quién está accediendo a sus datos.

La autentificación es un incordio. Tenemos dos métodos: uno fácil, OAuth 2, que sirve perfectamente cuando solo se desean realizar búsquedas sencillas, y otro complejo, OAuth 1, que es necesario cuando se quieren realizar acciones (por ejemplo, tuitear) o (para nuestro caso en particular) conectar con el *stream* de Twitter.

Así que nos quedamos con la forma más complicada, que trataremos de automatizar tanto como podamos.

En primer lugar, necesitamos la clave API y la clave secreta API (a veces conocida como clave de consumidor y clave secreta de consumidor, respectivamente). Obtendré la mía a partir de variables de entorno (no dude en poner las suyas si lo desea):

```
import os
```

```
# Cambie sin dudarlo sus claves directamente
CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY")
CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET")
```

Ahora podemos instanciar el cliente:

```
import webbrowser
from twython import Twython
# Obtiene cliente temporal para recuperar URL de autenticación
temp_client = Twython(CONSUMER_KEY, CONSUMER_SECRET)
temp_creds = temp_client.get_authentication_tokens()
url = temp_creds['auth_url']
# Ahora visita la URL para autorizar a la aplicación y obtener un PIN
print(f"go visit {url} and get the PIN code and paste it below")
webbrowser.open(url)
PIN_CODE = input("please enter the PIN code: ")
# Ahora usamos ese PIN_CODE para obtener los tokens reales
auth_client = Twython(CONSUMER_KEY,
                      CONSUMER_SECRET,
                      temp_creds['oauth_token'],
                      temp_creds['oauth_token_secret'])
final_step = auth_client.get_authorized_tokens(PIN_CODE)
ACCESS_TOKEN = final_step['oauth_token']
ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']
# Y obtiene una nueva instancia Twython usándolos.
twitter = Twython(CONSUMER_KEY,
                  CONSUMER_SECRET,
                  ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET)
```

Truco: Quizá en este momento le interese guardar `ACCESS_TOKEN` y `ACCESS_TOKEN_SECRET` en un sitio seguro, de modo que la próxima vez no tenga que pasar por todo este jaleo de nuevo.

En cuanto tengamos una instancia Twython autenticada, podemos empezar a realizar búsquedas:

```
# Busca tuits que contengan la frase "datascience"
for status in twitter.search(q='"data science"')["statuses"]:
   user = status["user"]["screen_name"]
   text = status["text"]
   print(f"{user}: {text}\n")
```

Si ejecutamos esto, obtendremos algunos tuits como estos:

```
haithemnyc: Data scientists with the technical savvy &amp;amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP
RPubsRecent: Data Science http://t.co/6hcHUz2PHM
spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization. So easy and Awesome.
```

Que no son muy interesantes, en parte porque la API Search de Twitter solo muestra el montón de resultados recientes que quiera. Cuando estamos haciendo ciencia de datos, solemos querer muchos tuits. Aquí es donde resulta de gran utilidad la API Streaming.[7] Permite conectarse al gran Firehose de Twitter (mejor dicho, a una pequeña parte). Para utilizarlo, será necesario autentificarse utilizando sus *tokens* de acceso.

Para acceder a la API Streaming con Twython, tenemos que definir una clase que herede de `TwythonStreamer` y que anule su método `on_success`, y posiblemente también su método `on_error`:

```
from twython import TwythonStreamer
# Añadir datos a una variable global es bastante pobre
# pero simplifica mucho el ejemplo
tweets = []
class MyStreamer(TwythonStreamer):
   def on_success(self, data):
      """
      What do we do when Twitter sends us data?
      Here data will be a Python dict representing a tweet.
      """
      # Solo queremos recopilar tuits en inglés
      if data.get('lang') == 'en':
         tweets.append(data)
         print(f"received tweet #{len(tweets)}")
      # Para cuando hemos recopilado bastantes
      if len(tweets) >= 100:
         self.disconnect()
   def on_error(self, status_code, data):
      print(status_code, data)
      self.disconnect()
```

`MyStreamer` conectará con el *stream* de Twitter y esperará a que Twitter le

pase datos. Cada vez que reciba datos (aquí, un tuit representado como un objeto de Python), los pasa al método `on_success`, que los añade a nuestra lista `tweets` si su idioma es inglés, y, una vez que ha recogido 1.000 tuits, se desconecta del *streamer*.

Todo lo que queda por hacer es inicializarlo y empezar a ejecutarlo:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
# empieza a consumir estados públicos que contienen 'data'
stream.statuses.filter(track='data')
# pero si queremos empezar a consumir una muestra de *todos* los estados
públicos
# stream.statuses.sample()
```

Esto se ejecutará hasta que recopile 100 tuits (o hasta que encuentre un error) y se detendrá, momento en el cual podremos empezar a analizar esos tuits. Por ejemplo, podríamos encontrar los *hashtags* más habituales con:

```
top_hashtags = Counter(hashtag['text'].lower()
                    for tweet in tweets
                    for hashtag in tweet["entities"]["hashtags"])
print(top_hashtags.most_common(5))
```

Cada tuit contiene muchos datos. Puede investigar por sí mismo o buscar en la documentación de las API de Twitter.[8]

Nota: En un proyecto real, probablemente no quiera confiar en una `list` en memoria para almacenar los tuits. Lo mejor que podría hacer sería guardarlos en un archivo o en una base de datos, de modo que así dispondría de ellos permanentemente.

Para saber más

- pandas, en `http://pandas.pydata.org/`, es la librería que utilizan normalmente los científicos de datos para trabajar con datos (específicamente, para importarlos).
- Scrapy, en `http://scrapy.org/`, es una librería repleta de funciones

para crear complicados raspadores web, que hagan cosas como seguir enlaces desconocidos.

- Kaggle, en https://www.kaggle.com/datasets, alberga una gran colección de conjuntos de datos.

---

[1] https://stackoverflow.com/questions/15587877/run-a-python-script-in-terminal-without-the-python-command.

[2] https://www.crummy.com/software/BeautifulSoup/.

[3] https://docs.python-requests.org/en/latest/.

[4] https://docs.github.com/es/rest.

[5] https://github.com/realpython/list-of-python-api-wrappers.

[6] https://github.com/ryanmcgrath/twython.

[7] https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data.

[8] https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object.

10 Trabajar con datos

Los expertos suelen poseer más datos que criterio.

—Colin Powell

Trabajar con datos es un arte, así como una ciencia. En general, hemos estado hablando de la parte científica, pero en este capítulo nos centraremos en el arte.

Explorar los datos

Una vez identificadas las preguntas que intentamos responder y después de haber obtenido datos, quizá se sienta tentado a meterse de lleno y empezar inmediatamente a crear modelos y obtener respuestas. Pero es necesario resistirse a este impulso. El primer paso debe ser explorar los datos.

Explorar datos unidimensionales

El caso más sencillo es tener un conjunto de datos unidimensional, que no es más que una colección de números. Por ejemplo, podría ser el número de minutos promedio al día que cada usuario se pasa en un sitio web, el número de veces que cada uno de los vídeos de tutoriales de ciencia de datos de una colección es visionado, o el número de páginas de cada uno de los libros de ciencia de datos que hay en una biblioteca.

Un primer paso obvio es calcular algunas estadísticas de resumen. Nos interesa saber cuántos puntos de datos tenemos, el menor, el mayor, la media y la desviación estándar.

Pero incluso estos datos no tienen por qué ofrecer un elevado nivel de comprensión. El siguiente paso correcto sería crear un histograma, en el que se agrupan los datos en *buckets* discretos y se cuenta cuántos puntos caen en

cada *bucket*:

```
from typing import List, Dict
from collections import Counter
import math
import matplotlib.pyplot as plt
def bucketize(point: float, bucket_size: float) -> float:
   """Floor the point to the next lower multiple of bucket_size"""
   return bucket_size * math.floor(point / bucket_size)
def make_histogram(points: List[float], bucket_size: float) -> Dict[float,
int]:
   """Buckets the points and counts how many in each bucket"""
   return Counter(bucketize(point, bucket_size) for point in points)
def plot_histogram(points: List[float], bucket_size: float, title: str = ""):
   histogram = make_histogram(points, bucket_size)
   plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
   plt.title(title)
```

Por ejemplo, tengamos en cuenta los dos siguientes conjuntos de datos:

```
import random
from scratch.probability import inverse_normal_cdf
random.seed(0)
# uniforme entre -100 y 100
uniform = [200 * random.random()–100 for _ in range(10000)]
# distribución normal con media 0, desviación estándar 57
normal = [57 * inverse_normal_cdf(random.random())
        for _ in range(10000)]
```

Ambos tienen medias próximas a 0 y desviaciones estándares cercanas a 58. Sin embargo, tienen distribuciones muy distintas. La figura 10.1 muestra la distribución de `uniform`:

```
plot_histogram(uniform, 10, “Uniform Histogram”)
```

Mientras que la figura 10.2 muestra la distribución de `normal`:

```
plot_histogram(normal, 10, “Normal Histogram”)
```

En este caso, las dos distribuciones tienen `max` y `min` bastante diferentes, pero ni siquiera saber esto habría sido suficiente para entender cómo difieren.

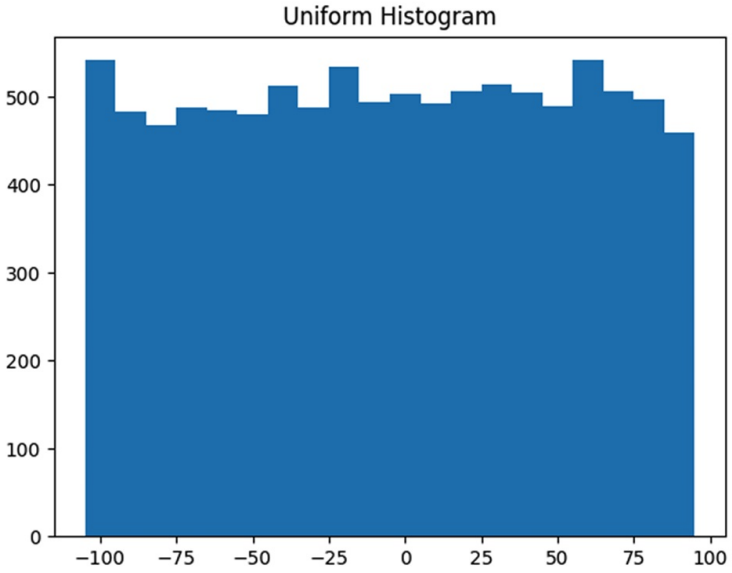


Figura 10.1. Histograma de `uniform`.

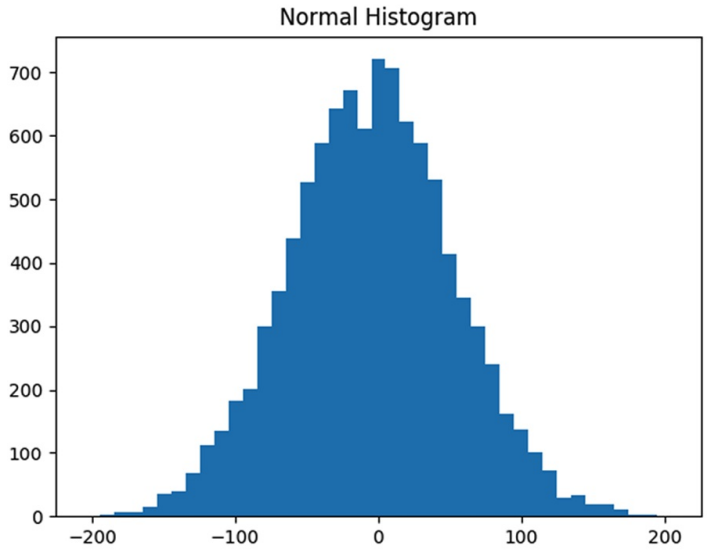


Figura 10.2. Histograma de `normal`.

Dos dimensiones

Ahora imaginemos que tenemos un conjunto de datos con dos

dimensiones. Quizá, además de los minutos diarios, tenemos años de experiencia en ciencia de datos. Por supuesto que queremos entender cada dimensión de manera individual, pero probablemente también nos interese dispersar los datos.

Por ejemplo, veamos otro conjunto de datos imaginario:

```
def random_normal() -> float:
   """Returns a random draw from a standard normal distribution"""
   return inverse_normal_cdf(random.random())
xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

Si ejecutáramos `plot_histogram` en `ys1` e `ys2`, obtendríamos trazados de aspecto similar (de hecho, ambos están distribuidos normalmente con la misma media y desviación estándar).

Pero cada uno tiene una distribución conjunta diferente con `xs`, como puede verse en la figura 10.3:

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()
```

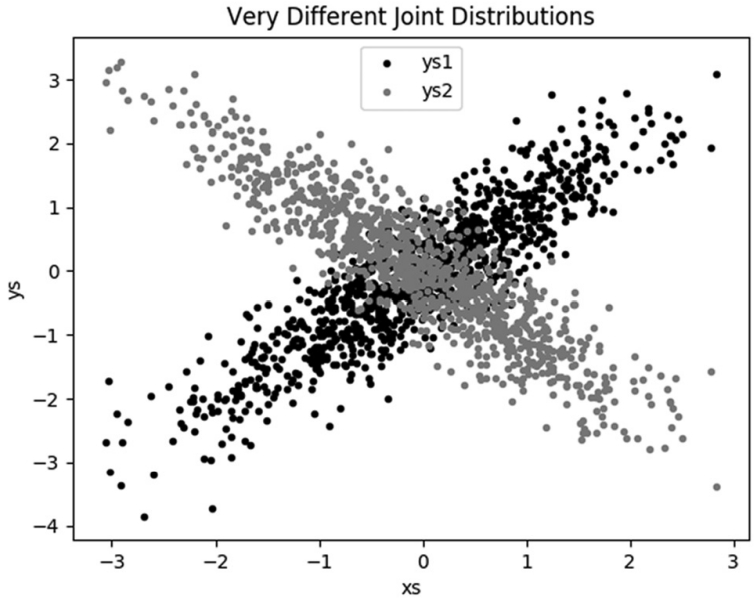


Figura 10.3. Dispersando dos `ys` distintos.

Esta diferencia también se haría patente si mirásemos las correlaciones:

```
from scratch.statistics import correlation
print(correlation(xs, ys1))      # más o menos 0.9
print(correlation(xs, ys2))      # más o menos -0.9
```

Muchas dimensiones

Si tenemos muchas dimensiones, nos interesará saber cómo se relacionan todas ellas entre sí. Una forma sencilla de averiguarlo es mirar la matriz de correlación, en la que la entrada de la fila *i* y la columna *j* es la correlación entre la dimensión *i* y la dimensión *j* de los datos:

```
from scratch.linear_algebra import Matrix, Vector, make_matrix
def correlation_matrix(data: List[Vector]) -> Matrix:
   """
   Returns the len(data) x len(data) matrix whose (i, j)-th entry
   is the correlation between data[i] and data[j]
   """
   def correlation_ij(i: int, j: int) -> float:
      return correlation(data[i], data[j])
   return make_matrix(len(data), len(data), correlation_ij)
```

Un método más visual (si no tenemos demasiadas dimensiones) es hacer una matriz de *scatterplot* o de diagrama de dispersión (figura 10.4), que muestre todos los gráficos de dispersión por pares. Para ello, utilizaremos `plt.subplots`, que nos permite crear *subplots* de nuestro gráfico. Le damos el número de filas y columnas y devuelve un objeto `figure` (que no usaremos) y un *array* bidimensional de objetos `axes` (que mostraremos en pantalla):

```
# corr_data es una lista de vectores de 100 dimensiones
num_vectors = len(corr_data)
fig, ax = plt.subplots(num_vectors, num_vectors)
for i in range(num_vectors):
   for j in range(num_vectors):
      # Dispersa column_j en el eje x frente a column_i en el eje y
      if i != j: ax[i][j].scatter(corr_data[j], corr_data[i])
      # a menos que i == j, en ese caso muestra el nombre de la serie
      else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
       xycoords='axes fraction',
       ha="center", va="center")
      # Luego oculta etiquetas de eje, salvo los diagramas izquierdo e inferior
      if i < num_vectors-1: ax[i][j].xaxis.set_visible(False)
      if j > 0: ax[i][j].yaxis.set_visible(False)
# Fija las etiquetas de ejes de abajo derecha y arriba izquierda, que están mal
# porque sus diagramas solo contienen texto
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())
plt.show()
```

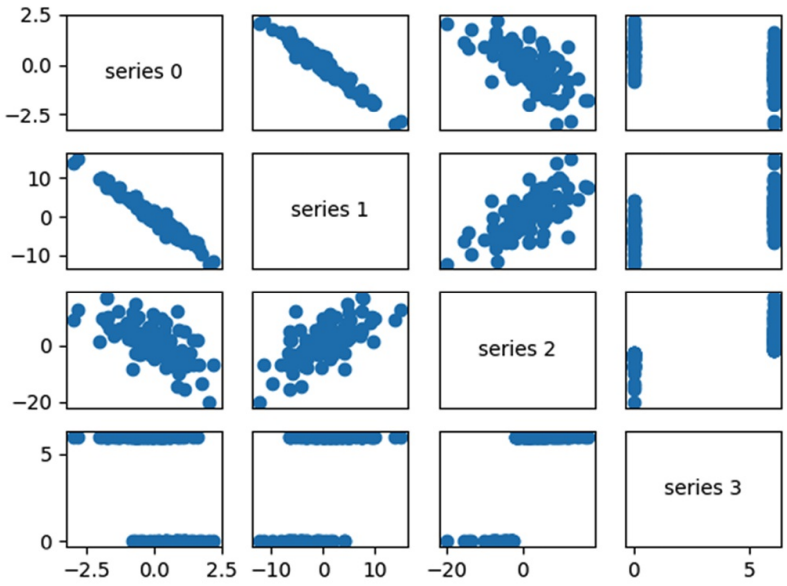


Figura 10.4. Matriz de diagrama de dispersión.

Mirando los diagramas de dispersión, podemos ver que la serie 1 está correlacionada muy negativamente con la serie 0, la serie 2 lo está positivamente con la serie 1 y la serie 3 solo toma los valores 0 y 6, correspondiendo el 0 a los valores pequeños de la serie 2 y el 6 a los valores grandes.

Esta es una forma rápida de hacerse una idea general de cuál de las variables está correlacionada (a menos que se pase horas retocando matplotlib para mostrar las cosas exactamente como las quiere, en cuyo caso no es un método rápido).

Utilizar NamedTuples

Una forma habitual de representar datos es utilizando `dict`:

```
import datetime
stock_price = {‘closing_price’: 102.06,
          ‘date’: datetime.date(2014, 8, 29),
          ‘symbol’: ‘AAPL’}
```

Hay varias razones por las que esto, sin embargo, no es lo ideal. Es una representación ligeramente ineficaz (un `dict` implica gastos extra), de modo que, si tenemos muchos precios de acciones, ocuparán más memoria de la necesaria. En general, esto es una consideración de menor importancia.

Un problema mayor es que acceder a las cosas mediante una clave `dict` tiene tendencia a producir errores. El siguiente código funcionará sin errores y solo hará lo incorrecto:

```
# huy, error
stock_price[‘cosing_price’] = 103.06
```

Finalmente, aunque podemos anotar los tipos de diccionarios uniformes:

```
prices: Dict[datetime.date, float] = {}
```

No hay un modo útil de anotar diccionarios como datos que tengan muchos tipos de valores distintos, de forma que también perdemos el poder de las comprobaciones de tipos.

Como alternativa, Python incluye una clase `namedtuple`, que es como una `tuple` pero con nombres:

```
from collections import namedtuple
StockPrice = namedtuple(‘StockPrice’, [‘symbol’, ‘date’, ‘closing_price’])
price = StockPrice(‘MSFT’, datetime.date(2018, 12, 14), 106.03)
assert price.symbol == ‘MSFT’
assert price.closing_price == 106.03
```

Como las `tuple` normales, las `namedtuple` son inmutables, lo que significa que no se pueden modificar sus valores una vez que se crean. De vez en cuando, esto se interpondrá en nuestro camino, pero en general es algo bueno.

Vemos que no hemos resuelto aún el tema de la anotación de tipo. Lo hacemos utilizando la variante con nombre, `NamedTuple`:

```
from typing import NamedTuple
class StockPrice(NamedTuple):
   symbol: str
   date: datetime.date
```

```
    closing_price: float
    def is_high_tech(self) -> bool:
        """It's a class, so we can add methods too"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)
assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

Ahora el editor nos puede ayudar, como muestra la figura 10.5.

Figura 10.5. Útil editor.

Nota: Muy poca gente utiliza así `NamedTuple`. ¡Pero deberían!

Clases de datos

Las clases de datos son (más o menos) una versión mutable de `NamedTuple` (digo "más o menos" porque las `NamedTuple` representan sus datos de manera compacta como tuples, mientras que las *dataclasses* son clases de Python normales que simplemente se encargan de generar automáticamente ciertos métodos).

Nota: Las clases de datos son nuevas en Python 3.7. Si está utilizando una versión más antigua, esta sección no le servirá.

La sintaxis es muy parecida a la de `NamedTuple`. Pero, en lugar de heredar de una clase base, utilizamos un decorador:

```
from dataclasses import dataclass
@dataclass
```

```
class StockPrice2:
   symbol: str
   date: datetime.date
   closing_price: float
   def is_high_tech(self) -> bool:
      """It's a class, so we can add methods too"""
      return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']
price2 = StockPrice2('MSFT', datetime.date(2018, 12, 14), 106.03)
assert price2.symbol == 'MSFT'
assert price2.closing_price == 106.03
assert price2.is_high_tech()
```

Como ya hemos dicho antes, la gran diferencia es que podemos modificar los valores de la instancia de una clase de datos:

```
# división de acciones
price2.closing_price /= 2
assert price2.closing_price == 51.03
```

Si intentáramos modificar un campo de la versión `NamedTuple`, obtendríamos un `AttributeError`.

Esto nos deja también susceptibles al tipo de errores que esperamos evitar no utilizando `dict`:

```
# Es una clase regular, así que añada campos siempre que quiera.
price2.cosing_price = 75                          # huy
```

No utilizaremos clases de datos, pero es fácil que se las encuentre por ahí fuera.

Limpiar y preparar datos

Los datos del mundo real están "sucios". Muchas veces tendremos que trabajar con ellos antes de poder utilizarlos. Hemos visto ejemplos en el capítulo 9. Hay que convertir cadenas de texto en `float` o `int` antes de poder usarlos, o revisar en busca de valores perdidos, valores atípicos (*outliers*) y datos erróneos.

Anteriormente, hicimos esto justo antes de usar los datos:

```
    closing_price = float(row[2])
```

Pero probablemente induce menos errores analizar una función que podemos probar:

```
from dateutil.parser import parse
def parse_row(row: List[str]) -> StockPrice:
   symbol, date, closing_price = row
   return StockPrice(symbol=symbol,
                date=parse(date).date(),
                closing_price=float(closing_price))
# A probar la función
stock = parse_row(["MSFT", "2018-12-14", "106.03"])
assert stock.symbol == "MSFT"
assert stock.date == datetime.date(2018, 12, 14)
assert stock.closing_price == 106.03
```

¿Qué pasa si hay datos erróneos? ¿O un valor “float” que en realidad no representa un número? ¿Quizá es mejor obtener un `None` que el programa se cuelgue?

```
from typing import Optional
import re
def try_parse_row(row: List[str]) -> Optional[StockPrice]:
   symbol, date_, closing_price_ = row
   # El símbolo de acción debe estar en mayúscula
   if not re.match(r"^[A-Z]+$", symbol):
      return None
   try:
      date = parse(date_).date()
   except ValueError:
      return None
   try:
      closing_price = float(closing_price_)
   except ValueError:
      return None
   return StockPrice(symbol, date, closing_price)
# Debería devolver None por errores
assert try_parse_row(["MSFT0", "2018-12-14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12—14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12-14", "x"]) is None
# Pero debería devolver lo mismo que antes si los datos son correctos
assert try_parse_row(["MSFT", "2018-12-14", "106.03"]) == stock
```

Por ejemplo, si tenemos precios de acciones delimitados por comas con datos sobrantes:

```
AAPL,6/20/2014,90.91
MSFT,6/20/2014,41.68
FB,6/20/3014,64.5
AAPL,6/19/2014,91.86
MSFT,6/19/2014,n/a
FB,6/19/2014,64.34
```

Ahora podemos leer y devolver solo las filas válidas:

```
import csv
data: List[StockPrice] = []
with open("comma_delimited_stock_prices.csv") as f:
   reader = csv.reader(f)
   for row in reader:
      maybe_stock = try_parse_row(row)
      if maybe_stock is None:
         print(f"skipping invalid row: {row}")
      else:
         data.append(maybe_stock)
```

Y decidir lo que queremos hacer con las no válidas. En general, las tres opciones son deshacerse de ellas, volver al origen y tratar de arreglar los datos sobrantes/faltantes, o no hacer nada y cruzar los dedos. Si hay una sola fila errónea de millones, probablemente no pasa nada por ignorarla. Pero, si la mitad de las filas tienen datos sobrantes, es algo que conviene arreglar.

Algo correcto que podemos hacer a continuación es buscar valores atípicos (*outliers*), utilizando las técnicas vistas en el apartado “Explorar datos” del comienzo de este capítulo o investigando como corresponde. Por ejemplo, ¿se dio cuenta de que una de las fechas del archivo de acciones tenía el año 3014? No tendría por qué dar error, pero es claramente incorrecto, y se obtendrían resultados caóticos si no se detectara. A los conjuntos de datos reales les faltan puntos decimales, tienen ceros de más, errores tipográficos y otros incontables problemas que nosotros tenemos que localizar (quizá este no sea oficialmente su trabajo, pero ¿quién más lo va a hacer?).

Manipular datos

Una de las habilidades más importantes de un científico de datos es la manipulación de los mismos. Se trata de un acercamiento general más que de una técnica específica, así que simplemente veremos unos cuantos ejemplos para que se haga una idea.

Imaginemos que tenemos un montón de datos de precios de acciones con este aspecto:

```
data = [
  StockPrice(symbol='MSFT',
         date=datetime.date(2018, 12, 24),
         closing_price=106.03),
  # ...
]
```

Empecemos por plantear preguntas sobre estos datos. Por el camino intentaremos observar patrones en lo que estamos haciendo y abstraer algunas herramientas para facilitar la manipulación.

Por ejemplo, supongamos que queremos conocer el precio de cierre máximo posible para AAPL. Dividamos esto en pasos:

1. Limitarnos a las filas AAPL.
2. Coger el `closing_price` de cada fila.
3. Tomar el `max` de esos precios.

Podemos hacer las tres cosas a la vez utilizando una lista de comprensión:

```
max_aapl_price = max(stock_price.closing_price
                for stock_price in data
                if stock_price.symbol == "AAPL")
```

De forma más general, nos podría interesar conocer el precio de cierre máximo posible para cada acción de nuestro conjunto de datos. Una forma de hacer esto es:

1. Crear un `dict` para mantener controlados los precios máximos (utilizaremos un `defaultdict` que devuelve menos infinito para valores que faltan, ya que cualquier precio será mayor).

2. Iterar nuestros datos, actualizándolos.

Este es el código:

```
from collections import defaultdict
max_prices: Dict[str, float] = defaultdict(lambda: float('-inf'))
for sp in data:
   symbol, closing_price = sp.symbol, sp.closing_price
   if closing_price > max_prices[symbol]:
      max_prices[symbol] = closing_price
```

Ahora podemos empezar a pedir cosas más complicadas, como averiguar cuáles son los cambios porcentuales de un día mayor y menor de nuestro conjunto de datos. El cambio porcentual es `price_today / price_yesterday -1`, lo que significa que necesitamos un modo de asociar el precio de hoy y el de ayer. Una forma es agrupando los precios por símbolo, y después, dentro de cada grupo:

1. Ordenar los precios por fecha.
2. Utilizar `zip` para obtener pares (anterior, actual).
3. Convertir los pares en nuevas filas de "cambio porcentual".

Empecemos agrupando los precios por símbolo:

```
from typing import List
from collections import defaultdict
# Recopila los precios por símbolo
prices: Dict[str, List[StockPrice]] = defaultdict(list)
for sp in data:
   prices[sp.symbol].append(sp)
```

Como los precios son tuplas, se clasifican por sus campos en orden: primero por símbolo, después por fecha y por último por precio. Esto significa que, si tenemos varios precios, todos con el mismo símbolo, `sort` los ordenará por fecha (y después por precio, lo que no hace nada, ya que tenemos solo uno por fecha), que es lo que queremos:

```
# Ordena los precios por fecha
```

```
prices = {symbol: sorted(symbol_prices)
        for symbol, symbol_prices in prices.items()}
```

Y que podemos utilizar para calcular una secuencia de cambios por día:

```
def pct_change(yesterday: StockPrice, today: StockPrice) -> float:
   return today.closing_price / yesterday.closing_price–1
class DailyChange(NamedTuple):
   symbol: str
   date: datetime.date
   pct_change: float
def day_over_day_changes(prices: List[StockPrice]) -> List[DailyChange]:
   """
   Assumes prices are for one stock and are in order
   """
   return [DailyChange(symbol=today.symbol,
                date=today.date,
                pct_change=pct_change(yesterday, today))
         for yesterday, today in zip(prices, prices[1:])]
```

Y recopilarlos después todos:

```
all_changes = [change
           for symbol_prices in prices.values()
           for change in day_over_day_changes(symbol_prices)]
```

Momento en el cual es fácil encontrar el mayor y el menor:

```
max_change = max(all_changes, key=lambda change: change.pct_change)
# ver p. ej. http://news.cnet.com/2100-1001-202143.html
assert max_change.symbol == 'AAPL'
assert max_change.date == datetime.date(1997, 8, 6)
assert 0.33 < max_change.pct_change < 0.34
min_change = min(all_changes, key=lambda change: change.pct_change)
# ver p.ej. http://money.cnn.com/2000/09/29/markets/techwrap/
assert min_change.symbol == ,AAPL'
assert min_change.date == datetime.date(2000, 9, 29)
assert -0.52 < min_change.pct_change < -0.51
```

Ahora se puede utilizar este nuevo conjunto de datos `all_changes` para averiguar qué mes es el mejor para invertir en acciones tecnológicas. Veremos el cambio diario medio por mes:

```
changes_by_month: List[DailyChange] = {month: [] for month in range(1, 13)}
for change in all_changes:
   changes_by_month[change.date.month].append(change)
avg_daily_change = {
   month: sum(change.pct_change for change in changes) / len(changes)
   for month, changes in changes_by_month.items()
}
# Octubre es el mejor mes
assert avg_daily_change[10] == max(avg_daily_change.values())
```

Estaremos haciendo este tipo de manipulaciones a lo largo del libro, normalmente sin llamar de una forma demasiado explícita la atención sobre ellas.

Redimensionar

Muchas técnicas son sensibles a la dimensión de los datos. Por ejemplo, imaginemos que tenemos un conjunto de datos que consiste en las alturas y pesos de cientos de científicos de datos, y que estamos tratando de identificar *clusters* de tamaños de cuerpos. De forma intuitiva, nos gustaría que los agrupamientos representaran puntos uno al lado del otro, lo que significa que necesitamos una cierta noción de distancia entre puntos. Ya tenemos la función euclidiana `distance`, de modo que la forma natural de hacer esto podría ser tratar pares (altura, peso) como puntos en un espacio bidimensional. Veamos las personas que aparecen en la tabla 10.1.

Tabla 10.1. Alturas y pesos.

Persona	Altura(pulgadas)	Altura (centímetros)	Peso (libras)
A	63	160	150
B	67	170,2	160
C	70	177,8	171

Si medimos la altura en pulgadas, entonces el vecino más próximo a B es A:

```
from scratch.linear_algebra import distance
a_to_b = distance([63, 150], [67, 160])         # 10.77
a_to_c = distance([63, 150], [70, 171])         # 22.14
b_to_c = distance([67, 160], [70, 171])         # 11.40
```

Pero, si medimos la altura en centímetros, sin embargo el vecino más próximo a B es C:

```
a_to_b = distance([160, 150], [170.2, 160])       # 14.28
a_to_c = distance([160, 150], [177.8, 171])       # 27.53
b_to_c = distance([170.2, 160], [177.8, 171])     # 13.37
```

Obviamente es un problema el hecho de que cambiar las unidades pueda cambiar así los resultados. Por esta razón, cuando las dimensiones no sean comparables una con otra, redimensionaremos en ocasiones nuestros datos de forma que cada dimensión tenga media 0 y desviación estándar 1. Así nos deshacemos efectivamente de las unidades, convirtiendo cada dimensión en "desviaciones estándares de la media".

Para empezar, tendremos que calcular `mean` y `standard_deviation` para cada posición:

```
from typing import Tuple
from scratch.linear_algebra import vector_mean
from scratch.statistics import standard_deviation
def scale(data: List[Vector]) -> Tuple[Vector, Vector]:
   """returns the mean and standard deviation for each position"""
   dim = len(data[0])
   means = vector_mean(data)
   stdevs = [standard_deviation([vector[i] for vector in data])
        for i in range(dim)]
return means, stdevs
vectors = [[-3, -1, 1], [-1, 0, 1], [1, 1, 1]]
means, stdevs = scale(vectors)
assert means == [-1, 0, 1]
assert stdevs == [2, 1, 0]
```

Después podemos emplearlas para crear un nuevo conjunto de datos:

```
def rescale(data: List[Vector]) -> List[Vector]:
   """
   Rescales the input data so that each position has
   mean 0 and standard deviation 1. (Leaves a position
   as is if its standard deviation is 0.)
   """
   dim = len(data[0])
   means, stdevs = scale(data)
   # Hace una copia de cada vector
   rescaled = [v[:] for v in data]
   for v in rescaled:
      for i in range(dim):
         if stdevs[i] > 0:
            v[i] = (v[i]–means[i]) / stdevs[i]
   return rescaled
```

Por supuesto, escribimos una prueba para comprobar que `rescale` hace lo que pensamos que hace:

```
means, stdevs = scale(rescale(vectors))
assert means == [0, 0, 1]
assert stdevs == [1, 1, 0]
```

Como siempre, necesitamos aplicar nuestro criterio. Si tomáramos un enorme conjunto de datos de alturas y pesos y lo filtráramos para quedarnos solo con las personas con alturas de entre 69,5 pulgadas y 70,5 pulgadas, es bastante probable (dependiendo de la pregunta que estemos tratando de responder) que la variación restante sea simplemente ruido, y quizá no queramos poner su desviación estándar al mismo nivel que las desviaciones de otras dimensiones.

Un inciso: tqdm

Con frecuencia, acabaremos haciendo cálculos que requieren mucho tiempo. Cuando estemos haciendo esto, nos gustará saber que estamos haciendo progresos y calcular el tiempo que se supone que tendremos que esperar.

Una forma de hacerlo es con la librería `tqdm`, que genera barras de progreso personalizadas. Lo utilizaremos un poco a lo largo del libro, de modo que aprovechemos la oportunidad que se nos brinda de aprender cómo funciona.

Lo primero es instalarlo:

```
python -m pip install tqdm
```

Solo hace falta conocer unas cuantas funciones. La primera es que un iterable envuelto en `tqdm.tqdm` producirá una barra de progreso:

```
import tqdm
for i in tqdm.tqdm(range(100)):
   # funciona algo lento
   _ = [random.random() for _ in range(1000000)]
```

Que produce un resultado parecido a este:

```
56%|
```

```
| 56/100 [00:08<00:06, 6.49it/s]
```

En particular, muestra qué fracción del bucle está terminada (aunque no se puede hacer esto si se utiliza un generador), cuánto tiempo se ha estado ejecutando y cuánto tiempo más espera hacerlo.

En este caso (donde simplemente estamos envolviendo una llamada a `range`), basta con utilizar `tqdm.range`.

También se puede configurar la descripción de la barra de progreso mientras está funcionando. Para ello, hay que capturar el iterador `tqdm` en una sentencia `with`:

```
from typing import List
def primes_up_to(n: int) -> List[int]:
   primes = [2]
   with tqdm.trange(3, n) as t:
      for i in t:
         # i es primo si ningún primo menor lo divide
         i_is_prime = not any(i % p == 0 for p in primes)
         if i_is_prime:
```

```
            primes.append(i)
        t.set_description(f"{len(primes)} primes")
    return primes
my_primes = primes_up_to(100_000)
```

Esto añade una descripción como la siguiente, con un contador que se actualiza cuando se descubren nuevos primos:

```
5116 primes: 50%|
```

```
| 49529/99997 [00:03<00:03, 15905.90it/s]
```

Utilizar `tqdm` puede hacer que el código inspire desconfianza (ya que, a veces, la pantalla se redibuja pésimamente, y otras veces el bucle directamente se colgará). Si envolvemos accidentalmente un bucle `tqdm` dentro de otro, podrían ocurrir cosas raras. Lo normal es que sus beneficios superen ampliamente estos inconvenientes, así que trataremos de utilizarlo siempre que tengamos entre manos cálculos de ejecución lenta.

Reducción de dimensionalidad

En ocasiones, las dimensiones “reales” (o útiles) de los datos podrían no corresponder con las dimensiones que tenemos. Por ejemplo, veamos el conjunto de datos representado en la figura 10.6.

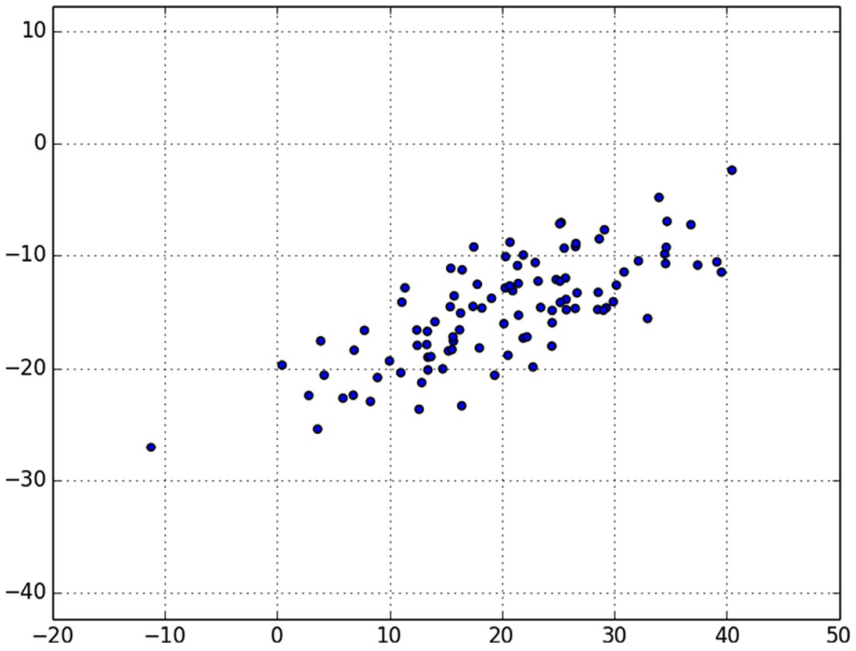


Figura 10.6. Datos con los ejes “erróneos”.

La mayor parte de la variación de los datos parece producirse en una única dimensión, que no corresponde con el eje x o y.

Cuando ocurre esto, podemos utilizar una técnica denominada análisis de componentes principales o PCA (*Principal Component Analysis*) para extraer una o más dimensiones que capturen tanto de la variación de datos como sea posible.

Nota: En la práctica, no utilizaríamos esta técnica en un conjunto de datos de tan baja dimensión. Principalmente, la reducción de dimensionalidad es útil cuando el conjunto de datos tiene un gran número de dimensiones y deseamos encontrar un pequeño subconjunto que capture la mayor parte de la variación. Lamentablemente, esta situación es difícil de ilustrar en un libro en formato de dos dimensiones.

Como primer paso a dar, tendremos que traducir los datos de modo que cada dimensión tenga media 0:

```
from scratch.linear_algebra import subtract
def de_mean(data: List[Vector]) -> List[Vector]:
   """Recenters the data to have mean 0 in every dimension"""
```

```
    mean = vector_mean(data)
    return [subtract(vector, mean) for vector in data]
```

(Si no hacemos esto, es probable que nuestras técnicas identifiquen la propia media en lugar de la variación en los datos).

La figura 10.7 muestra los datos de ejemplo tras aplicar media 0.

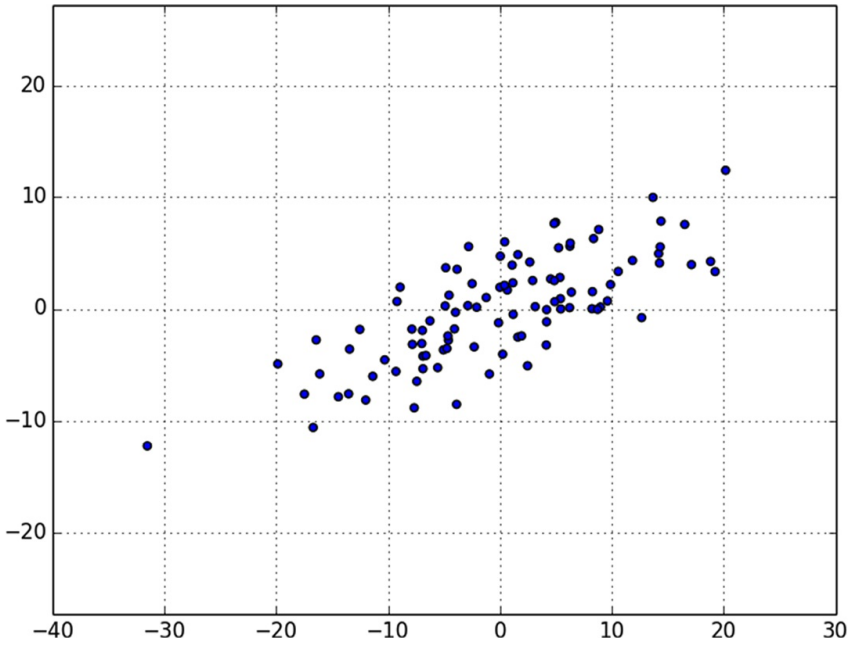


Figura 10.7. Datos tras aplicar media 0.

Ahora, dada una matriz *X* con media 0, podemos preguntar cuál es la dirección que captura la mayor varianza en los datos.

Específicamente, dada una dirección `d` (un vector de magnitud 1), cada fila `x` de la matriz extiende `dot(x, d)` en la dirección `d`. Y cada vector no cero `w` determina una dirección si lo redimensionamos para que tenga una magnitud 1:

```
from scratch.linear_algebra import magnitude
def direction(w: Vector) -> Vector:
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

De esta forma, dado un vector no cero `w`, podemos calcular la varianza de

nuestro conjunto de datos en la dirección determinada por `w`:

```
from scratch.linear_algebra import dot
def directional_variance(data: List[Vector], w: Vector) -> float:
   """
   Returns the variance of x in the direction of w
   """
   w_dir = direction(w)
   return sum(dot(v, w_dir) ** 2 for v in data)
```

Nos gustaría encontrar la dirección que maximice esta varianza. Podemos hacerlo utilizando el descenso de gradiente, tan pronto como tengamos la función gradiente:

```
def directional_variance_gradient(data: List[Vector], w: Vector) -> Vector:
   """
   The gradient of directional variance with respect to w
   """
   w_dir = direction(w)
   return [sum(2 * dot(v, w_dir) * v[i] for v in data)
        for i in range(len(w))]
```

Y ahora el principal componente que tenemos es simplemente la dirección que maximiza la función `directional_variance`:

```
from scratch.gradient_descent import gradient_step
def first_principal_component(data: List[Vector],
      n: int = 100,
      step_size: float = 0.1) -> Vector:
   # Inicia con una suposición al azar
   guess = [1.0 for _ in data[0]]
   with tqdm.trange(n) as t:
      for _ in t:
         dv = directional_variance(data, guess)
         gradient = directional_variance_gradient(data, guess)
         guess = gradient_step(guess, gradient, step_size)
         t.set_description(f"dv: {dv:.3f}")
   return direction(guess)
```

En el conjunto de datos con media 0, esto devuelve la dirección `[0.924, 0.383]`, que parece capturar el eje principal a lo largo del cual varían nuestros

datos (figura 10.8).

Una vez localizada la dirección que es el primer componente principal, podemos proyectar en ella nuestros datos para hallar los valores de dicho componente:

```
from scratch.linear_algebra import scalar_multiply
def project(v: Vector, w: Vector) -> Vector:
   """return the projection of v onto the direction w"""
   projection_length = dot(v, w)
   return scalar_multiply(projection_length, w)
```

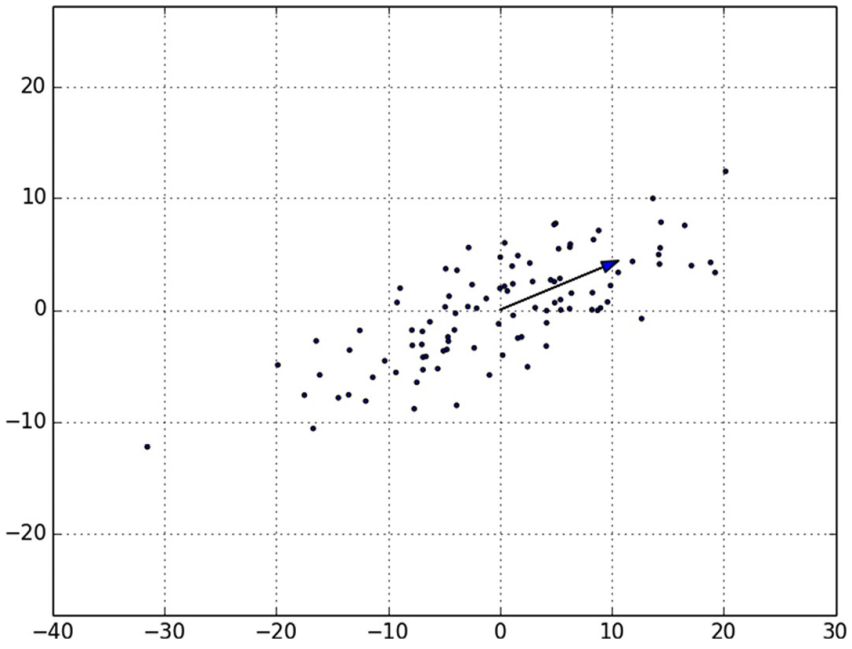


Figura 10.8. Primer componente principal.

Si queremos hallar más componentes, primero eliminamos las proyecciones de los datos:

```
from scratch.linear_algebra import subtract
def remove_projection_from_vector(v: Vector, w: Vector) -> Vector:
   """projects v onto w and subtracts the result from v"""
   return subtract(v, project(v, w))
def remove_projection(data: List[Vector], w: Vector) -> List[Vector]:
   return [remove_projection_from_vector(v, w) for v in data]
```

Como este conjunto de datos de ejemplo es solo bidimensional, tras

eliminar el primer componente, lo que queda será, efectivamente, unidimensional (figura 10.9).

En este momento, podemos hallar el siguiente componente principal repitiendo el proceso con el resultado de `remove_projection` (figura 10.10).

En un conjunto de datos de muchas dimensiones, podemos encontrar de forma iterativa tantos componentes como queramos:

```
def pca(data: List[Vector], num_components: int) -> List[Vector]:
   components: List[Vector] = []
   for _ in range(num_components):
      component = first_principal_component(data)
      components.append(component)
      data = remove_projection(data, component)
   return components
```

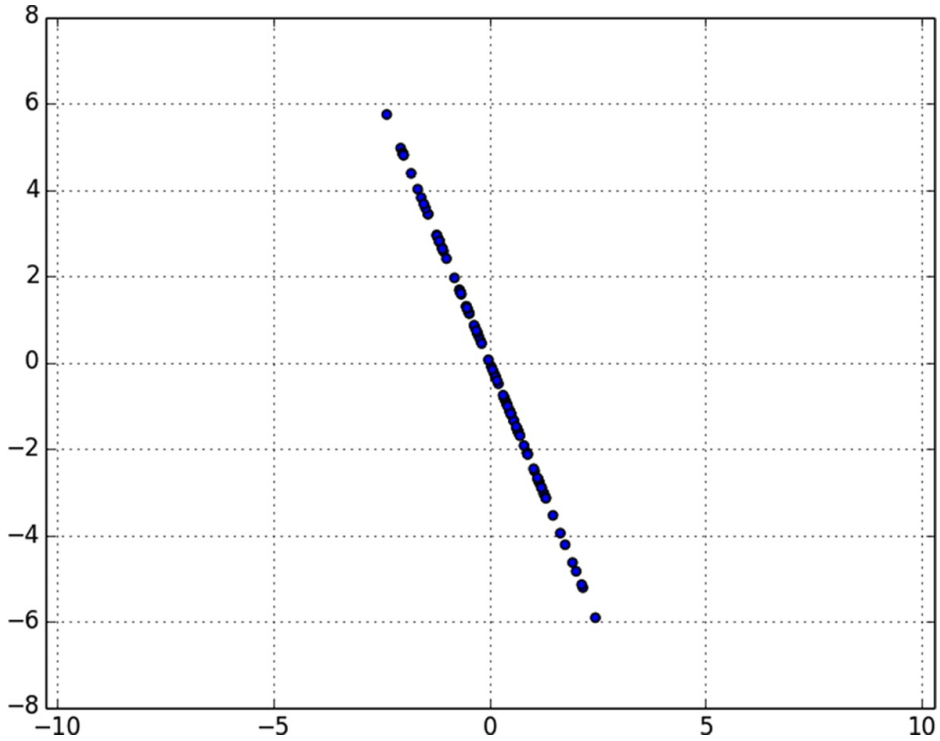


Figura 10.9. Datos tras eliminar el primer componente principal.

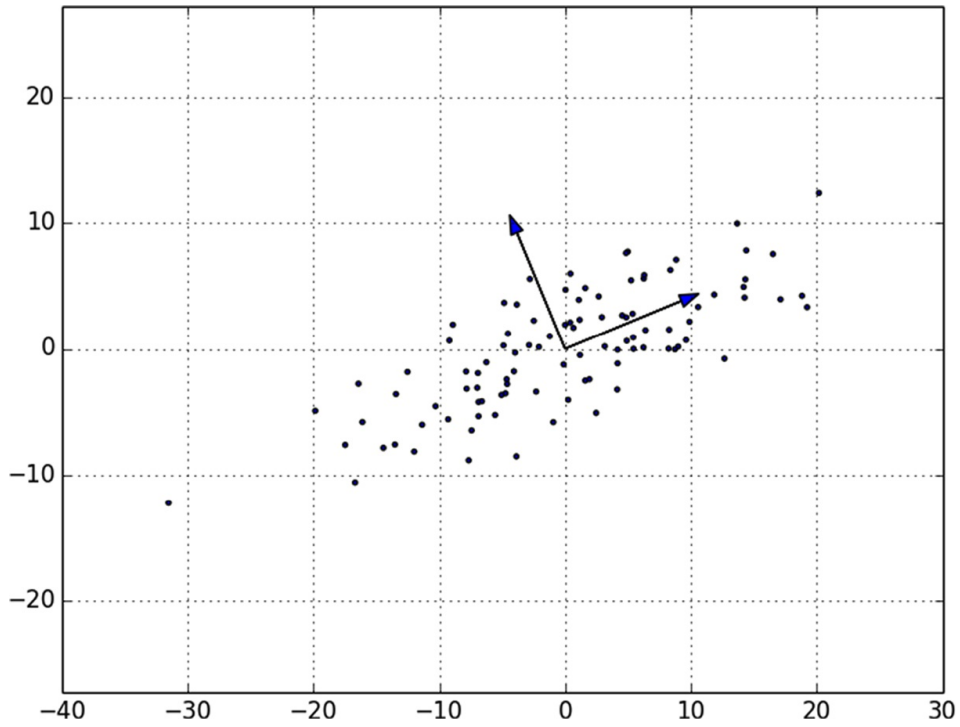


Figura 10.10. Primeros dos componentes principales.

Podemos después transformar nuestros datos en el espacio de menos dimensiones atravesado por los componentes:

```
def transform_vector(v: Vector, components: List[Vector]) -> Vector:
   return [dot(v, w) for w in components]
def transform(data: List[Vector], components: List[Vector]) -> List[Vector]:
   return [transform_vector(v, components) for v in data]
```

Esta técnica es válida por varias razones. Primero, puede permitirnos limpiar nuestros datos eliminando las dimensiones de ruido y consolidando las dimensiones altamente correlacionadas.

Segundo, tras extraer una representación de baja dimensión de nuestros datos, podemos utilizar diversas técnicas que no funcionen tan bien en datos de alta dimensión. Veremos ejemplos de dichas técnicas a lo largo del libro.

Al mismo tiempo, mientras esta técnica puede ayudarnos a crear modelos mejores, también puede hacer que esos modelos sean más difíciles de interpretar. Es fácil entender conclusiones como “cada año adicional de experiencia suma una media de 10.000 euros al salario”, pero es mucho más difícil dar sentido a “cada incremento de 0,1 en el tercer componente

principal suma una media de 10.000 euros al salario”.

Para saber más

- Como mencionamos al final del capítulo 9, pandas, en `http://pandas.pydata.org/`, es probablemente la herramienta principal de Python para limpiar, preparar, manipular y trabajar con datos. Todos los ejemplos que hicimos a mano en este capítulo podrían haberse hecho de un modo mucho más sencillo utilizando pandas. *Python for Data Analysis* (O’Reilly), de Wes McKinney, es probablemente la mejor manera de aprender pandas.
- scikit-learn tiene una amplia variedad de funciones de descomposición de matrices, en `https://scikit-learn.org/stable/modules/classes.html#module-sklearn.decomposition`, incluyendo PCA.

11 Machine learning (aprendizaje automático)

Siempre estoy dispuesto a aprender, aunque no siempre me gusta que me den lecciones.

—Winston Churchill

Mucha gente piensa que la ciencia de datos es más que nada aprendizaje automático o *machine learning*, y que los científicos de datos se pasan el día creando, entrenando y modificando modelos de *machine learning* (aunque, pensándolo bien, muchas de esas personas no saben ni siquiera lo que es esto realmente). En realidad, lo que hace la ciencia de datos es convertir problemas de empresa en problemas de datos, y recoger, comprender, limpiar y formatear datos, tras de lo cual el aprendizaje automático es casi una anécdota. Puede que lo sea, pero es interesante y esencial, y merece mucho la pena conocerlo para poder hacer ciencia de datos.

Modelos

Antes de poder hablar de *machine learning*, tenemos que hablar primero de los modelos.

¿Qué es un modelo? No es más que la especificación de una relación matemática (o probabilística) existente entre distintas variables.

Por ejemplo, si la idea es recaudar dinero para una red social, sería interesante crear un modelo de negocio (probablemente en una hoja de cálculo) que admitiera entradas como “número de usuarios”, “ingresos de publicidad por usuario” y “número de empleados” y obtuviera el beneficio anual para los siguientes años. Una receta de un libro de cocina conlleva un modelo que relaciona entradas como “número de comensales” y “hambre” con las cantidades de ingredientes necesarias. En las partidas de póker que

pueden verse en televisión, la “probabilidad de victoria” de cada jugador se estima en tiempo real basándose en un modelo que tiene en cuenta las cartas que se han levantado hasta el momento y la distribución de cartas de la baraja.

Probablemente, el modelo de negocio está basado en sencillas relaciones matemáticas: el beneficio es el ingreso menos el gasto, el ingreso es igual a las unidades vendidas multiplicadas por el precio medio, etc.

El modelo de receta se basa casi seguro en el método de prueba y error (alguien fue a una cocina y probó distintas combinaciones de ingredientes hasta que encontró una que le gustó), mientras que el modelo de póker tiene como base la teoría de la probabilidad, las reglas del póker y ciertas suposiciones razonablemente inocuas sobre el proceso aleatorio mediante el cual se reparten las cartas.

¿Qué es el machine learning?

Cada uno tiene su propia definición exacta, pero utilizaremos *machine learning* para referirnos a la creación y utilización de modelos que se aprenden a partir de los datos. En otros contextos se podría denominar modelado predictivo o minería de datos, pero nos quedaremos con aprendizaje automático o *machine learning*. Habitualmente, nuestro objetivo será utilizar datos ya existentes para desarrollar modelos que podamos emplear para predecir resultados diversos con datos nuevos, como por ejemplo:

- Si un email es *spam* o no.
- Si una transacción con tarjeta de crédito es fraudulenta.
- En qué anuncio es más probable que un comprador haga clic.
- Qué equipo de fútbol va a ganar la Champions.

Veremos modelos supervisados (en los que hay un conjunto de datos etiquetado con las respuestas correctas de las que aprender) y modelos no supervisados (que no tienen tales etiquetas). Hay otros tipos, como

semisupervisados (en los que solo parte de los datos están etiquetados), modelos *online* (en los que el modelo necesita ser continuamente ajustado con los datos más recientes) y modelos por refuerzo (en los que, tras realizar una serie de predicciones, el modelo obtiene una señal indicando lo bien que lo hizo), que no trataremos en este libro. Pero, hasta en la situación más simple, hay universos enteros de modelos que podrían describir la relación en la que estamos interesados. En la mayoría de los casos, nosotros mismos elegiremos una familia parametrizada de modelos, y después utilizaremos los datos para aprender parámetros que de algún modo son óptimos.

Por ejemplo, podríamos suponer que la altura de una persona es (más o menos) una función lineal de su peso, y utilizar después los datos para saber cuál es esa función lineal. O también podríamos creer que un árbol de decisión es una buena forma de diagnosticar cuáles son las enfermedades que tienen nuestros pacientes y utilizarlo luego para descubrir cuál sería un árbol así "óptimo". A lo largo del resto del libro, investigaremos distintas familias de modelos que podemos aprender.

Pero, antes de poder hacer esto, tenemos que comprender mejor los fundamentos del *machine learning*. En el resto de este capítulo hablaremos de algunos de estos conceptos básicos, antes de pasar a los modelos propiamente dichos.

Sobreajuste y subajuste

Un peligro habitual en *machine learning* es el sobreajuste u *overfitting* (producir un modelo que funcione bien con los datos con los que se le entrena, pero que generaliza muy mal con datos nuevos). Podría implicar descubrir ruido en los datos. También podría suponer aprender a identificar determinadas entradas en lugar de cualesquiera factores que sean realmente predictivos para el resultado deseado.

La otra cara de esto es el subajuste o *underfitting* (producir un modelo que no funcione bien ni siquiera con los datos de entrenamiento; aunque, normalmente, cuando esto ocurre, uno mismo decide que el modelo no es lo bastante bueno y sigue buscando uno mejor).

En la figura 11.1 he ajustado tres polinomios a una muestra de datos (no se preocupe por cómo lo he hecho, llegaremos a ello en capítulos posteriores).

La línea horizontal muestra el mejor polinomio ajustado de grado 0 (es decir, constante), que subajusta enormemente los datos de entrenamiento. El mejor polinomio ajustado de grado 9 (es decir, de parámetro 10) pasa exactamente por cada uno de los puntos de datos de entrenamiento, pero lo sobreajusta en gran medida; si tuviéramos que elegir algunos puntos de datos más, muy probablemente los perdería por mucho. Y la línea de grado 1 alcanza un buen equilibrio; está bastante cerca de cada punto, y (si estos datos son representativos) la línea estará probablemente cerca también de nuevos puntos de datos.

Está claro que los modelos que son demasiado complejos llevan al sobreajuste y no generalizan bien más allá de los datos con los que fueron entrenados. Así que ¿cómo nos aseguramos de que nuestros modelos no son demasiado complejos? La estrategia más básica conlleva utilizar datos diferentes para entrenar y probar el modelo.

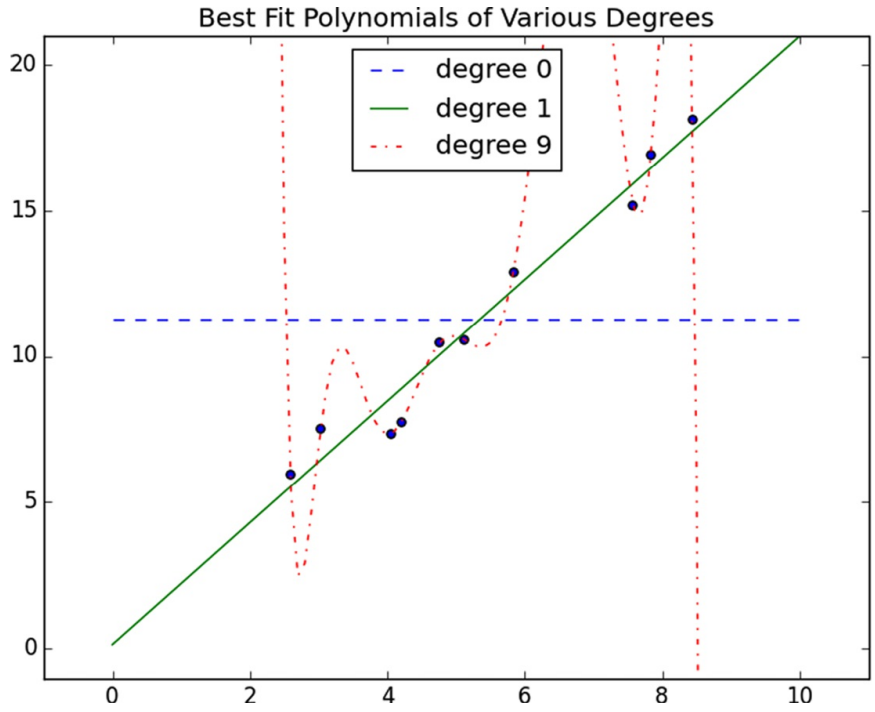


Figura 11.1. Sobreajuste y subajuste.

La forma más sencilla de hacer esto es dividir el conjunto de datos, de forma que (por ejemplo) dos tercios de él se utilicen para entrenar el modelo, después de lo cual medimos el rendimiento del modelo en el tercio restante:

```
import random
from typing import TypeVar, List, Tuple
X = TypeVar('X')              # tipo genérico para representar un punto de
                              datos
def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
   """Split data into fractions [prob, 1–prob]"""
   data = data[:]                 # Hace una copia rápida
   random.shuffle(data)           # porque shuffle modifica la lista.
   cut = int(len(data) * prob)    # Usa prob para hallar un límite
   return data[:cut],             # y divide allí la lista mezclada.
   data[cut:]
data = [n for n in range(1000)]
train, test = split_data(data, 0.75)
# Las proporciones deberían ser correctas
assert len(train) == 750
assert len(test) == 250
# Y los datos originales deberían preservarse (en un cierto orden)
assert sorted(train + test) == data
```

Con frecuencia tendremos pares de variables de entrada y salida. En ese caso, debemos asegurarnos de poner juntos los valores correspondientes en los datos de entrenamiento o en los de prueba:

```
Y = TypeVar('Y')                # tipo genérico para representar variables de
                                salida
def train_test_split(xs: List[X],
                     ys: List[Y],
                     test_pct: float) -> Tuple[List[X], List[X], List[Y],
                                           List[Y]]:
   # Genera los índices y los divide
   idxs = [i for i in range(len(xs))]
   train_idxs, test_idxs = split_data(idxs, 1–test_pct)
   return ([xs[i] for i in              # x_train
   train_idxs],
      [xs[i] for i in test_idxs],       # x_test
      [ys[i] for i in train_idxs],      # y_train
      [ys[i] for i in test_idxs])       # y_test
```

Como siempre, queremos asegurarnos de que nuestro código funcione

bien:

```
xs = [x for x in range(1000)]      # xs son 1 ... 1000
ys = [2 * x for x in xs]           # cada y_i es el doble de x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)
# Revisa proporciones correctas
assert len(x_train) == len(y_train) == 750
assert len(x_test) == len(y_test) == 250
# Revisa puntos de datos correspondientes bien emparejados
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))
```

Tras de lo cual se puede hacer algo como:

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

Si el modelo estuviera sobreajustado a los datos de entrenamiento, entonces sería de esperar que funcionara mal en los datos de prueba (que son completamente distintos). Dicho de otro modo, si funciona bien en los datos de prueba, entonces podemos estar más seguros de que está ajustado en lugar de sobreajustado. Sin embargo, hay varias maneras en las que esto puede salir mal.

La primera es si hay patrones comunes en los datos de entrenamiento y de prueba que no generalizarían a un conjunto de datos más grande.

Por ejemplo, imaginemos que el conjunto de datos que tenemos consiste en actividad de usuario, con una fila por usuario y semana. En tal caso, la mayoría de los usuarios aparecerán en los datos de entrenamiento y en los de prueba, y determinados modelos podrían aprender a identificar usuarios en lugar de descubrir relaciones que impliquen atributos. En realidad, no es una gran preocupación, aunque me ocurrió una vez.

Un problema más importante es si se utiliza la separación prueba/entrenamiento no solo para juzgar un modelo, sino también para elegir entre muchos modelos. En ese caso, aunque cada modelo individual pueda no estar sobreajustado, “elegir un modelo que funcione mejor en el conjunto de prueba” es un metaentrenamiento que hace que el conjunto de prueba funcione como un segundo conjunto de entrenamiento (por supuesto,

el modelo que funcionó mejor en el conjunto de prueba va a funcionar bien en el de entrenamiento).

En una situación como esta, deberíamos dividir los datos en tres partes: un conjunto de entrenamiento para crear modelos, un conjunto de validación para elegir entre modelos entrenados y un conjunto de prueba para juzgar el modelo final.

Exactitud

Cuando no estoy haciendo ciencia de datos, hago incursiones en medicina. En mi tiempo libre he creado una prueba barata y no invasiva que se le puede dar a un recién nacido y que predice (con más de un 98 % de exactitud) si el recién nacido desarrollará leucemia. Mi abogado me ha convencido de que la prueba no se puede patentar, de modo que compartiré aquí los detalles: predice la leucemia si y solo si el bebé se llama Luke (que suena parecido a "*leukemia*", como se dice leucemia en inglés).

Como podemos comprobar, esta prueba tiene claramente más del 98 % de precisión. Sin embargo, es increíblemente ridícula, y es también una buena forma de ilustrar la razón por la cual no solemos utilizar el término "exactitud" para medir lo bueno que es un modelo (de clasificación binaria).

Supongamos que creamos un modelo para emitir un juicio binario. ¿Es este email *spam*? ¿Deberíamos contratar a este candidato? ¿Es uno de los viajeros de este avión un terrorista en secreto?

Dado un conjunto de datos etiquetados y un modelo predictivo como este, cada punto de datos está incluido en una de cuatro categorías:

Verdadero positivo
"Este mensaje es *spam*, y hemos predicho correctamente que lo es".
Falso positivo (error tipo 1)
"Este mensaje no es *spam*, pero hemos predicho que lo es".
Falso negativo (error tipo 2)
"Este mensaje es *spam*, pero hemos predicho que no lo es".
Verdadero negativo

“Este mensaje no es *spam*, y hemos predicho correctamente que no lo es”.

A menudo representamos estas categorías como contadores de una matriz de confusión:

	Es *spam*	**No es** *spam*
Predice "es *spam*"	Verdadero positivo	Falso positivo
Predice "no es *spam*"	Falso negativo	Verdadero negativo

Veamos cómo encaja mi prueba de la leucemia en esta estructura. En estos días, aproximadamente 5 bebés de cada 1.000 se llaman Luke,[1] y la prevalencia de la leucemia a lo largo de la vida es más o menos del 1,4 %, es decir, 14 personas de cada 1.000.[2]

Si pensamos que estos factores son independientes y aplicamos mi prueba “Luke viene de leucemia” a un millón de personas, esperaríamos ver una matriz de confusión como esta:

	Leucemia	**No leucemia**	**Total**
"Luke"	70	4.930	5.000
No "Luke"	13.930	981.070	995.000
Total	14.000	986.000	1.000.000

Podemos entonces utilizar estas matrices para calcular distintas estadísticas sobre el rendimiento de modelos. Por ejemplo, exactitud se define como la fracción de las predicciones correctas:

```
def accuracy(tp: int, fp: int, fn: int, tn: int) -> float:
   correct = tp + tn
   total = tp + fp + fn + tn
   return correct / total
assert accuracy(70, 4930, 13930, 981070) == 0.98114
```

Parece un número que impresiona bastante. Pero sin duda no es una buena prueba, lo que significa que probablemente no deberíamos creer demasiado en la exactitud pura y dura.

Es habitual mirar la combinación de precisión y recuerdo. La precisión mide lo exactas que fueron nuestras predicciones positivas:

```
def precision(tp: int, fp: int, fn: int, tn: int) -> float:
   return tp / (tp + fp)
assert precision(70, 4930, 13930, 981070) == 0.014
```

Y el recuerdo mide la fracción de los positivos que identificó nuestro modelo:

```
def recall(tp: int, fp: int, fn: int, tn: int) -> float:
   return tp / (tp + fn)
assert recall(70, 4930, 13930, 981070) == 0.005
```

Ambos son números terribles, que reflejan que es un modelo espantoso.

En ocasiones, la precisión y el recuerdo se combinan en la puntuación F1 (*F1 score*), que se define como:

```
def f1_score(tp: int, fp: int, fn: int, tn: int) -> float:
   p = precision(tp, fp, fn, tn)
   r = recall(tp, fp, fn, tn)
   return 2 * p * r / (p + r)
```

Esta es la media armónica[3] de precisión y recuerdo, y se sitúa forzosamente entre ellos. Normalmente, la elección de un modelo implica un término medio entre precisión y recuerdo. Un modelo que predice “sí” en cuanto tiene la más mínima confianza en este resultado tendrá probablemente un elevado recuerdo, pero una precisión baja; un modelo que prediga “sí” solo cuando tenga toda la confianza en que será así es probable que tenga un bajo recuerdo y una elevada precisión.

También se puede pensar en esto como en un término medio entre falsos positivos y falsos negativos. Decir “sí” con demasiada frecuencia dará muchos falsos positivos, pero decir “no” proporcionará muchos falsos negativos.

Supongamos que hubiera 10 factores de riesgo para la leucemia y que, cuántos más de ellos se tuvieran, más alta sería la probabilidad de padecer la enfermedad. En ese caso podemos imaginar un continuo de pruebas: “predecir leucemia si al menos hay un factor de riesgo”, “predecir leucemia si al menos hay dos factores de riesgo”, etc. A medida que aumenta el umbral, sube la precisión de la prueba (ya que es más probable que las personas con más factores de riesgo desarrollen la enfermedad), y disminuirá el recuerdo de la prueba (ya que cada vez menos posibles sufridores de la enfermedad alcanzarán el umbral). En casos así, elegir el umbral correcto es cuestión de encontrar el término medio adecuado.

El término medio entre sesgo y varianza

Otra forma de pensar con respecto al problema del sobreajuste es como en un término medio entre sesgo y varianza. Ambas son medidas de lo que ocurriría si hubiera que volver a entrenar nuestro modelo muchas veces en distintos conjuntos de datos de entrenamiento (a partir de la misma población más grande). Por ejemplo, el modelo de grado 0 de la sección anterior “Sobreajuste y subajuste” producirá muchos errores con prácticamente cualquier conjunto de entrenamiento (dibujado a partir de la misma población), lo que significa que tiene un alto sesgo. No obstante, cualesquiera dos conjuntos de entrenamiento aleatoriamente elegidos darían modelos bastante similares (ya que deberían tener valores promedio también bastante similares). De modo que decimos que tiene una varianza baja. El alto sesgo y la baja varianza corresponden normalmente al subajuste.

Por otro lado, el modelo de grado 9 encaja perfectamente en el conjunto de entrenamiento. Tiene muy bajo sesgo, pero muy alta varianza (ya que cualesquiera dos conjuntos de entrenamiento darían probablemente lugar a modelos muy diferentes). Esto corresponde al sobreajuste.

Pensar en problemas de modelos de este modo puede permitirnos averiguar qué hacer cuando el modelo no funciona tan bien.

Si el modelo tiene un elevado sesgo (lo que significa que funciona mal incluso con los datos de entrenamiento), algo que se puede probar es añadir

más funciones. Pasar del modelo de grado 0 de “Sobreajuste y subajuste” al modelo de grado 1 supuso una gran mejora. Si nuestro modelo tiene una alta varianza, se pueden eliminar funciones de forma similar. Pero otra solución es obtener más datos (si se puede).

En la figura 11.2, ajustamos un polinomio de grado 9 a muestras de distinto tamaño. El ajuste de modelo basado en 10 puntos de datos está por todas partes, como ya vimos antes. Pero, si entrenamos con 100 puntos de datos, hay mucho menos subajuste.

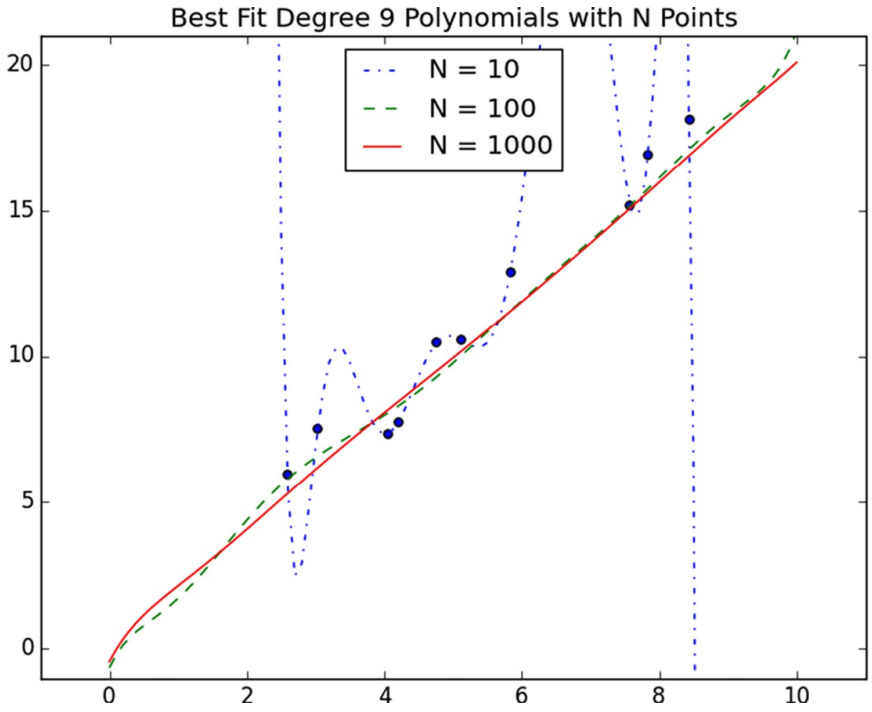


Figura 11.2. Reducir la varianza con más datos.

Y el modelo entrenado con 1.000 puntos de datos parece muy similar al modelo de grado 1. Manteniendo la complejidad del modelo constante, cuantos más datos tengamos, más difícil será que haya sobreajuste. Por otro lado, más datos no ayudarán con el sesgo. Si el modelo no utiliza suficientes funciones para capturar uniformidades en los datos, lanzarle más datos no servirá de nada.

Extracción y selección de características

Como ya se ha mencionado, cuando los datos no tienen suficientes características, es probable que el modelo subajuste; y, cuando los datos tienen demasiadas características, es fácil que sobreajuste. Pero ¿qué son las características, y de dónde proceden?

Las características son las entradas que le proporcionamos a nuestro modelo.

En el caso más sencillo, las características simplemente nos vienen dadas. Si queremos predecir el salario de alguien basándonos en sus años de experiencia, entonces los años de experiencia son la única característica que tenemos (aunque, como ya vimos en el apartado "Sobreajuste y subajuste", también se podría considerar añadir años de experiencia al cuadrado, al cubo, etc., si ello nos permitiera construir un modelo mejor).

Las cosas se ponen más interesantes a medida que los datos se van complicando. Supongamos que intentamos crear un filtro de *spam* para predecir si un email es basura o no. La mayoría de los modelos no sabrán qué hacer con un email sin depurar, que no es más que una colección de texto. Tendremos que extraer características. Por ejemplo:

- ¿Contiene el email la palabra viagra?
- ¿Cuántas veces aparece la letra d?
- ¿Cuál era el dominio del remitente?

La respuesta a la primera pregunta es simplemente sí o no, lo que codificaríamos normalmente como 1 o 0. La segunda es un número. Y la tercera es una elección entre un reducido conjunto de opciones.

Prácticamente siempre extraeremos características de nuestros datos que entran en una de estas tres categorías. Es más, los tipos de características que tenemos limitan los tipos de modelos que podemos utilizar.

- El clasificador Naive Bayes que crearemos en el capítulo 13 es adecuado para características de tipo sí o no, como la primera pregunta de la lista anterior.
- Los modelos de regresión, que estudiaremos en los capítulos 14 y 16, requieren características numéricas (que podrían incluir variables ficticias que son ceros y unos).

- Y los árboles de decisión, que veremos en el capítulo 17, pueden admitir datos numéricos o categóricos.

Aunque en el ejemplo de filtro de *spam* vimos formas de crear características, otras veces también veremos maneras de eliminarlas.

Por ejemplo, las entradas podrían ser vectores de varios cientos de números. Dependiendo de la situación, podría ser apropiado reducirlos a un puñado de dimensiones importantes (como en la sección "Reducción de la dimensionalidad" del capítulo 10) y utilizar solamente ese pequeño número de características. O podría ser adecuado utilizar una técnica (como la regularización, que veremos en la sección del mismo nombre del capítulo 15) que penaliza los modelos cuantas más características utilizan.

¿Cómo elegimos las características? Aquí entra en juego una combinación entre experiencia y conocimientos del sector. Si ha recibido muchos emails, entonces probablemente intuirá que la presencia de determinadas palabras podría ser un buen indicador de *spam*. Quizá también sea capaz de intuir que el número de letras "d" casi seguro que no es un buen indicador de *spam*. Pero, en general, siempre habrá que probar distintas cosas, lo que es parte de la diversión.

Para saber más

- ¡Siga leyendo! Los siguientes capítulos hablan de distintas familias de modelos de *machine learning*.
- El curso de *machine learning* de Coursera, en `https://www.coursera.org/learn/machine-learning`, es el MOOC original y un buen punto de partida para obtener una profunda comprensión de los fundamentos de *machine learning*.
- *The Elements of Statistical Learning*, de Jerome H. Friedman, Robert Tibshirani y Trevor Hastle (Springer), es un libro de texto un poco canónico que se puede descargar en línea gratuitamente en `https://hastie.su.domains/ElemStatLearn/printings/ESLII_prin`
  Pero queda avisado: es muy técnico en matemáticas.

[1] https://www.babycenter.com/baby-names-luke-2918.htm.

[2] https://seer.cancer.gov/statfacts/html/leuks.html.

[3] https://es.wikipedia.org/wiki/Media_arm%C3%B3nica.

12 k vecinos más cercanos

Si quieres molestar a tus vecinos, di la verdad sobre ellos.

—Pietro Aretino

Supongamos que alguien intenta predecir cómo voy a votar en las próximas elecciones. Si ese alguien no sabe nada sobre mí (y dispone de los datos), un planteamiento razonable podría ser ver cómo están pensando votar mis vecinos. Viviendo en Seattle, como yo, la intención de mis vecinos sin duda alguna es votar al candidato demócrata, lo que sugiere que "candidato demócrata" es asimismo una buena suposición para mí.

Ahora supongamos que esa persona sabe más sobre mí que solamente geografía; quizá conoce mi edad, mis ingresos, cuántos hijos tengo, etc. En la medida en que mi comportamiento se vea influido (o caracterizado) por esos conocimientos, parece probable que, de entre todas esas dimensiones, ver qué piensan los vecinos que están cerca de mí sea un indicador aún mejor que incluir a todos mis vecinos. Esta es la idea del método de clasificación de vecinos más cercanos.

El modelo

Vecinos más cercanos es uno de los modelos predictivos más sencillos que hay. No realiza suposiciones matemáticas y no exige ningún tipo de maquinaria pesada. Lo único que requiere es:

- Una cierta noción de distancia.
- La suposición de que los puntos que están cerca uno de otro son similares.

La mayor parte de las técnicas que veremos en este libro tienen en cuenta el conjunto de datos como un todo para descubrir patrones en los datos.

Vecinos más cercanos, por otra parte, descuida de forma consciente mucha información, ya que la predicción para cada nuevo punto depende únicamente del montón de puntos más cercanos a él.

Es más, probablemente vecinos más cercanos no ayudará a comprender los elementos causantes del fenómeno que se esté observando. Predecir mi voto basándose en el de mis vecinos no dice mucho sobre las causas que me hacen votar como lo hago, mientras que otro modelo alternativo que prediga mi voto basándose en (por ejemplo) mis ingresos y mi estado civil sí podría ayudar a averiguarlas.

En situaciones generales, tenemos puntos de datos y el correspondiente conjunto de etiquetas. Las etiquetas podrían ser `True` y `False`, indicando que cada entrada satisface una determinada condición, como "¿es *spam*?", "¿es venenoso?" o "¿sería divertido de ver?". O bien podrían ser categorías, como calificaciones de películas (G, PG, PG-13, R, NC-17). También podrían ser los nombres de los candidatos presidenciales o incluso lenguajes de programación favoritos.

En nuestro caso, los puntos de datos serán vectores, lo que significa que podemos utilizar la función `distance` del capítulo 4.

Supongamos que elegimos un número *k*, como 3 o 5. Entonces, cuando queremos clasificar un punto de datos nuevo, encontramos los *k* puntos más cercanos etiquetados y les dejamos votar en el nuevo resultado.

Para ello, necesitaremos una función que cuente votos. Una posibilidad es:

```
from typing import List
from collections import Counter
def raw_majority_vote(labels: List[str]) -> str:
   votes = Counter(labels)
   winner, _ = votes.most_common(1)[0]
   return winner
assert raw_majority_vote([‘a’, ‘b’, ‘c’, ‘b’]) == ‘b’
```

Pero esto no hace nada inteligente con los empates. Por ejemplo, imaginemos que estamos calificando películas y las cinco más cercanas están clasificadas como G, G, PG, PG y R. Vemos que tanto G como PG tienen dos votos. En ese caso, tenemos varias opciones:

- Elegir uno de los ganadores aleatoriamente.
- Ponderar los votos por distancia y elegir el ganador resultante.
- Reducir *k* hasta encontrar un ganador único.

Implementaremos la tercera opción:

```
def majority_vote(labels: List[str]) -> str:
   """Assumes that labels are ordered from nearest to farthest."""
   vote_counts = Counter(labels)
   winner, winner_count = vote_counts.most_common(1)[0]
   num_winners = len([count
                for count in vote_counts.values()
                if count == winner_count])
   if num_winners == 1:
      return winner                         # ganador único, así que lo devuelve
   else:
      return                                # prueba de nuevo sin el más lejano
      majority_vote(labels[:-1])
# Empate, así que busca los primeros 4, entonces 'b'
assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

Este método seguro que terminará funcionando, ya que en el peor de los casos lo iremos reduciendo todo hasta que quede una sola etiqueta, momento en el cual esa es la que gana.

Con esta función, es fácil crear un clasificador:

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance
class LabeledPoint(NamedTuple):
   point: Vector
   label: str
def knn_classify(k: int,
             labeled_points: List[LabeledPoint],
             new_point: Vector) -> str:
   # Ordena los puntos etiquetados de más cercano a más lejano.
   by_distance = sorted(labeled_points,
                  key=lambda lp: distance(lp.point, new_point))
   # Halla las etiquetas para los k más cercanos
   k_nearest_labels = [lp.label for lp in by_distance[:k]]
   # y les deja votar.
   return majority_vote(k_nearest_labels)
```

Veamos cómo funciona esto.

Ejemplo: el conjunto de datos iris

El conjunto de datos iris es una de las bases de *machine learning*. Contiene un grupo de medidas para 150 flores que representan tres especies de iris. Para cada flor tenemos la longitud y la anchura del pétalo, la longitud del sépalo, además de su especie. Se puede descargar en la página `https://archive.ics.uci.edu/ml/datasets/iris`.

```
import requests
data = requests.get(
   "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
)
with open('iris.dat', 'w') as f:
   f.write(data.text)
```

Los datos están separados por comas, con los campos:

```
sepal_length, sepal_width, petal_length, petal_width, class
```

Por ejemplo, la primera fila es algo así:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

En esta sección trataremos de crear un modelo que pueda predecir la clase (es decir, la especie) a partir de las cuatro primeras medidas.

Para empezar, carguemos y exploremos los datos. Nuestra función vecinos más cercanos espera un `LabeledPoint`, de modo que representemos nuestros datos de ese modo:

```
from typing import Dict
import csv
from collections import defaultdict
def parse_iris_row(row: List[str]) -> LabeledPoint:
   """
   sepal_length, sepal_width, petal_length, petal_width, class
   """
   measurements = [float(value) for value in row[:-1]]
```

```
    # la clase es p. ej. "Iris-virginica"; solo queremos "virginica"
    label = row[-1].split("-")[-1]
    return LabeledPoint(measurements, label)
with open('iris.data') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader]
# Agrupamos también solo los puntos por especie/etiqueta para trazarlos
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

Nos vendría bien trazar las medidas de forma que podamos ver cómo varían por especie. Lamentablemente, son cuatridimensionales, lo que hace que resulten difíciles de representar. Una cosa que podemos hacer es recurrir a los gráficos de dispersión para cada uno de los seis pares de medidas (figura 12.1). No explicaré todos los detalles, pero es una buena forma de ilustrar cosas más complicadas que se pueden hacer con matplotlib, por lo que vale la pena estudiarlo:

```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x']          # we have 3 classes, so 3 markers
fig, ax = plt.subplots(2, 3)
for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
        ax[row][col].set_title(f"{metrics[i]} vs {metrics[j]}", fontsize=8)
        ax[row][col].set_xticks([])
        ax[row][col].set_yticks([])
        for mark, (species, points) in zip(marks, points_by_species.items()):
            xs = [point[i] for point in points]
            ys = [point[j] for point in points]
            ax[row][col].scatter(xs, ys, marker=mark, label=species)
ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

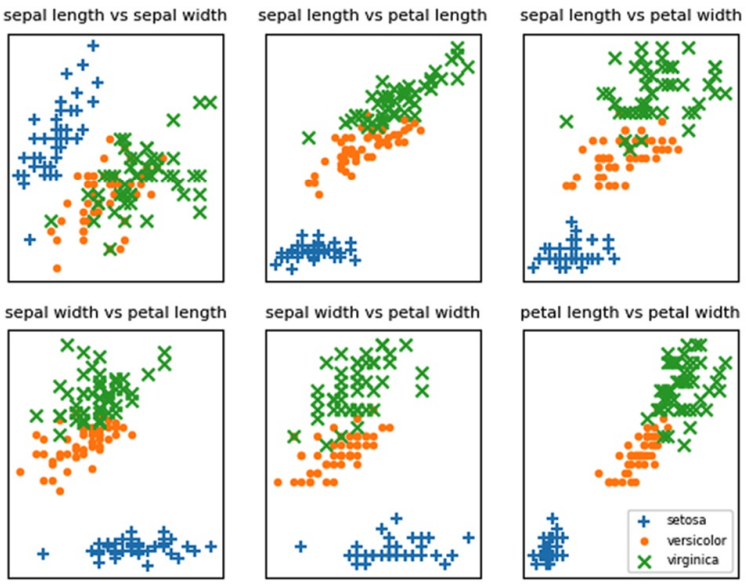


Figura 12.1. Gráficos de dispersión de iris.

Si echamos un vistazo a estos gráficos, parece que en realidad las medidas se agrupan por especie. Por ejemplo, mirando solamente la longitud y la anchura del sépalo, probablemente no se podría distinguir entre versicolor y virginica. Pero, en cuanto se añade a la mezcla la longitud y la anchura del pétalo, parece posible predecir la especie basándose en los vecinos más cercanos.

Para empezar, dividamos los datos en un conjunto de prueba y otro de entrenamiento:

```
import random
from scratch.machine_learning import split_data
random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
```

El conjunto de entrenamiento será los “vecinos” que utilizaremos para clasificar los puntos del conjunto de prueba. Simplemente tenemos que elegir un valor para *k*, el número de vecinos que consiguen votar. Demasiado pequeño (pensemos en $k = 1$), y dejaremos que los valores atípicos (*outliers*)

tengan demasiada influencia; demasiado grande (digamos $k = 105$), y simplemente predeciremos la clase más común del conjunto de datos.

En una aplicación real (y con más datos), podríamos crear un conjunto de validación diferente y emplearlo para elegir k. Aquí utilizaremos $k = 5$:

```
from typing import Tuple
# controla las veces que vemos (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0
for iris in iris_test:
   predicted = knn_classify(5, iris_train, iris.point)
   actual = iris.label
   if predicted == actual:
      num_correct += 1
   confusion_matrix[(predicted, actual)] += 1
pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

En este sencillo conjunto de datos, el modelo predice casi perfectamente. Hay una única versicolor para la que predice virginica, pero por lo demás acierta de pleno.

La maldición de la dimensionalidad

El algoritmo de k vecinos más cercanos da problemas con muchas dimensiones gracias a la "maldición de la dimensionalidad", que se reduce al hecho de que los espacios de muchas dimensiones son inmensos. Los puntos en los espacios con muchas dimensiones tienden a no estar en absoluto cerca unos de otros. Una forma de comprobar esto es generando de forma aleatoria pares de puntos en el "cubo unitario" d-dimensional en diversas dimensiones, y calculando las distancias entre ellos.

A estas alturas, generar puntos aleatorios ya debería ser algo natural:

```
def random_point(dim: int) -> Vector:
   return [random.random() for _ in range(dim)]
```

Igual que lo es escribir una función para generar las distancias:

```
def random_distances(dim: int, num_pairs: int) -> List[float]:
   return [distance(random_point(dim), random_point(dim))
      for _ in range(num_pairs)]
```

Para cada dimensión de 1 a 100, calcularemos 10.000 distancias, que usaremos para calcular la distancia media entre puntos y la distancia mínima entre puntos en cada dimensión (figura 12.2):

```
import tqdm
dimensions = range(1, 101)
avg_distances = []
min_distances = []
random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
   distances = random_distances(dim, 10000)         # 10.000 pares aleatorios
   avg_distances.append(sum(distances) / 10000)     # controla la media
   min_distances.append(min(distances))             # controla la mínima
```

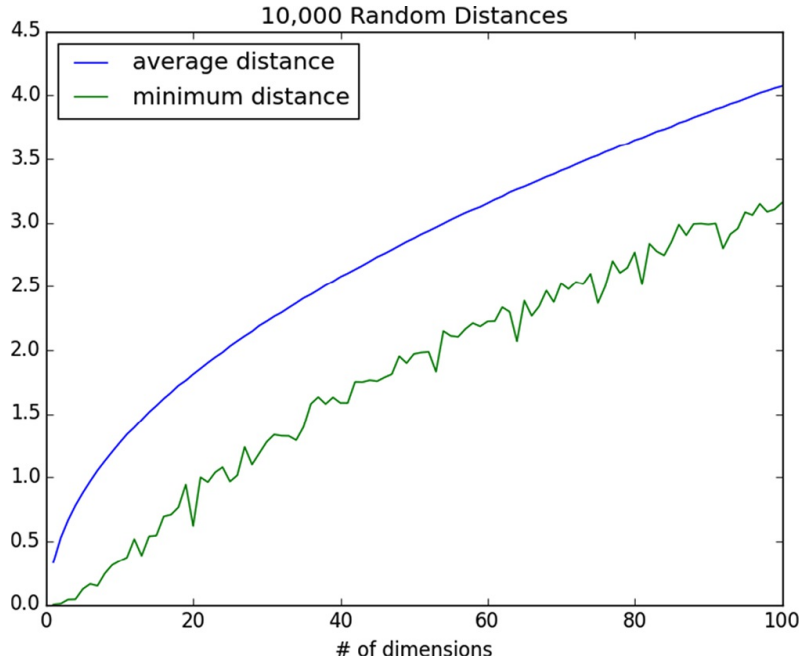


Figura 12.2. La maldición de la dimensionalidad.

A medida que aumenta el número de dimensiones, se incrementa también la distancia media entre puntos. Pero lo más problemático es la proporción

entre la distancia más cercana y la distancia media (figura 12.3):

```
min_avg_ratio = [min_dist / avg_dist
             for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

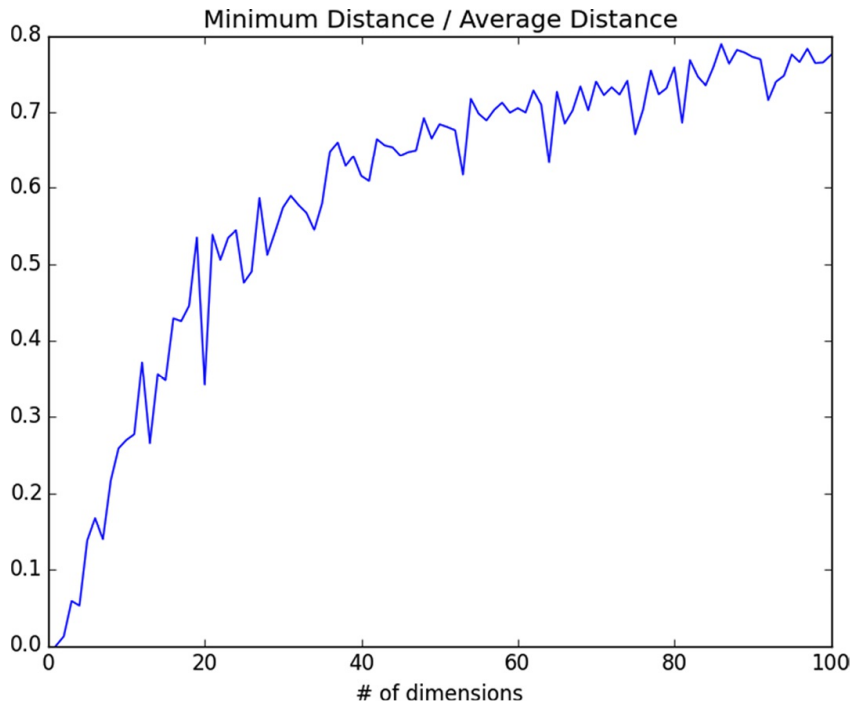


Figura 12.3. Otra vez la maldición de la dimensionalidad.

En conjuntos de datos de pocas dimensiones, los puntos más cercanos tienden a estar mucho más cerca que la media. Pero dos puntos están cerca solo si lo están en cada dimensión, y cada dimensión adicional (incluso aunque sea solo ruido) es otra oportunidad para que cada punto esté más lejos de los demás. Cuando se tienen muchas dimensiones, es probable que los puntos más próximos no estén mucho más cerca que la media, por tanto, que dos puntos estén cerca no significa mucho (a menos que haya mucha estructura en los datos que les haga comportarse como si fueran muy poco dimensionales).

Una forma diferente de pensar acerca de este problema tiene que ver con la escasez de espacios de muchas dimensiones.

Si elegimos 50 números aleatorios entre 0 y 1, probablemente obtendremos una muestra bastante buena del intervalo de la unidad (figura 12.4).

Si elegimos 50 puntos aleatorios dentro del cuadrado unitario, tendremos menos cobertura (figura 12.5).

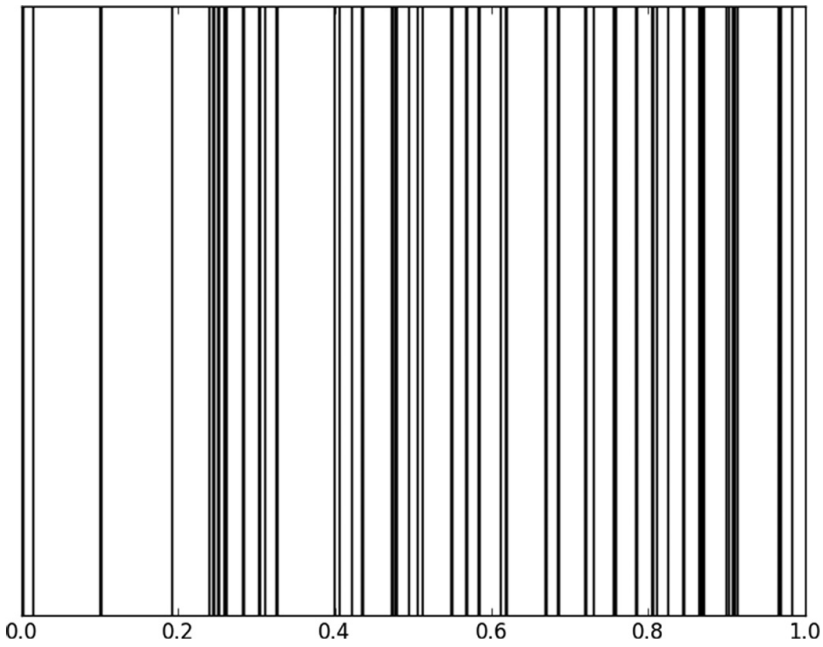


Figura 12.4. Cincuenta puntos aleatorios en una sola dimensión.

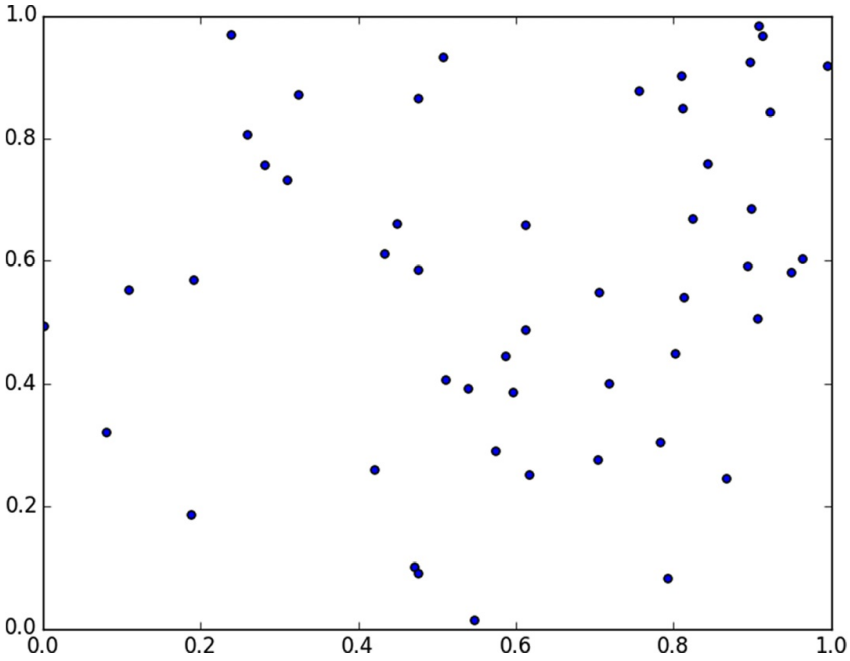


Figura 12.5. Cincuenta puntos aleatorios en dos dimensiones.

Y en tres dimensiones, todavía menos (figura 12.6).

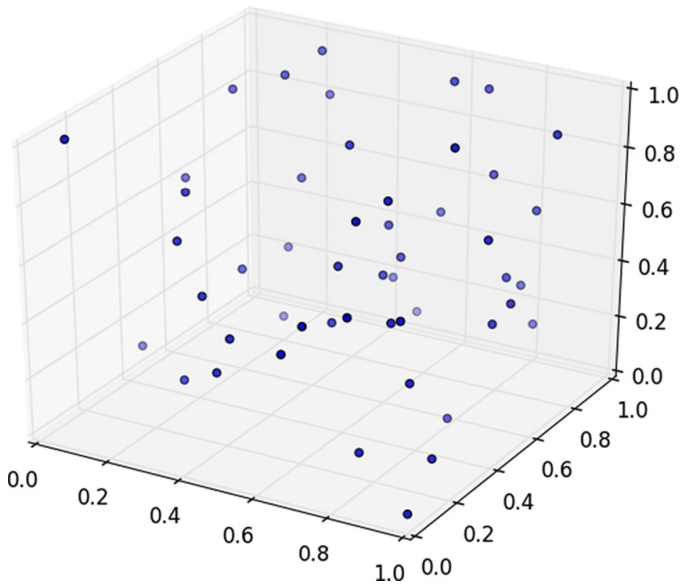


Figura 12.6. Cincuenta puntos aleatorios en tres dimensiones.

matplotlib no crea bien gráficos de cuatro dimensiones, de modo que hasta ahí llegaremos, pero ya se puede verificar que están empezando a haber grandes espacios vacíos sin puntos cerca. En más dimensiones (a menos que obtengamos más datos exponencialmente), esos grandes espacios vacíos representan regiones alejadas de todos los puntos que deseamos utilizar en nuestras predicciones.

Así que, si queremos utilizar vecinos más cercanos en muchas dimensiones, probablemente es una buena idea hacer primero algún tipo de reducción de dimensionalidad.

Para saber más

- scikit-learn tiene muchos modelos de vecinos más cercanos, en `https://scikit-learn.org/stable/modules/neighbors.html`.

13 Naive Bayes

Está bien ser ingenuo de corazón, pero no serlo de mente.

—Anatole France

Una red social no es muy buena si la gente no puede conectar. Según esto, DataSciencester tiene una característica muy popular que permite a sus miembros enviar mensajes a otros miembros. Aunque la mayoría de los miembros son ciudadanos responsables que solo envían mensajes agradables del tipo "¿cómo te va?", hay unos cuantos malandrines que envían correo basura de forma persistente sobre maneras de hacerse rico, productos farmacéuticos sin receta y programas de acreditación en ciencia de datos con ánimo de lucro. Los usuarios han empezado a quejarse, por lo que el vicepresidente de Mensajería le ha pedido que utilice la ciencia de datos para encontrar una manera de filtrar estos mensajes de *spam*.

Un filtro de spam realmente tonto

Supongamos un "universo" que consista en recibir un mensaje elegido aleatoriamente del total de posibles mensajes. Digamos que S es el evento "el mensaje es spam" y B el evento "el mensaje contiene la palabra bitcoin". El teorema de Bayes nos dice que la probabilidad de que el mensaje sea *spam*, condicionado a que contenga la palabra "bitcoin", es:

$$P(S|B) = [P(B|S)P(S)]/[P(B|S)P(S) + P(B|\neg S)P(\neg S)]$$

El numerador es la probabilidad de que un mensaje sea *spam* y contenga "bitcoin", mientras que el denominador es solo la probabilidad de que un mensaje contenga "bitcoin". Por lo tanto, se puede pensar en este cálculo como en la sencilla representación de la proporción de mensajes de bitcoin que son *spam*.

Si tenemos una gran colección de mensajes que sabemos que son *spam*, y otra gran colección de mensajes que sabemos que no son *spam*, podemos estimar fácilmente $P(B|S)$ y $P(B|\neg S)$. Si seguimos suponiendo que es igualmente probable que un determinado mensaje sea *spam* o no *spam* (de forma que $P(S) = P(\neg S) = 0{,}5$), entonces:

$$P(S|B) = P(B|S)/[P(B|S) + P(B|\neg S)$$

Por ejemplo, si el 50 % de los mensajes de *spam* tienen la palabra "bitcoin", pero solo el 1 % de los mensajes que no son *spam* la tienen, entonces la probabilidad de que cualquier email determinado que contenga la palabra "bitcoin" sea *spam* es:

$$0{,}5 / (0{,}5 + 0{,}01) = 98\ \%$$

Un filtro de spam más sofisticado

Supongamos ahora que tenemos un vocabulario formado por muchas palabras w_1, ..., w_n. Para llevar esto al reino de la teoría de la probabilidad, denominaremos X_i al evento "un mensaje contiene la palabra w_i". Vamos a imaginar que (mediante algún proceso no especificado en este momento) hemos hallado una estimación $P(X_i|S)$ para la probabilidad de que un mensaje de *spam* contenga la palabra *i*-ésima, y una estimación $P(X_i|\neg S)$ para la probabilidad de que un mensaje que no sea *spam* contenga la palabra *i*-ésima.

La clave de Naive Bayes es hacer la (gran) suposición de que las presencias (o ausencias) de cada palabra son independientes una de otra, condicionado todo ello a que un mensaje sea *spam* o no lo sea. Intuitivamente, esta suposición significa que saber si un determinado mensaje de *spam* contiene la palabra "bitcoin" no da ninguna información sobre si el mismo mensaje contiene la palabra Rolex. En términos matemáticos, esto significa que:

$$P(X_1 = x_1, ..., X_n = x_n|S) = P(X_1 = x_1|S) \text{ x } ... \text{ x } P(X_n = x_n|S)$$

Esto es una suposición extrema (hay una razón por la que esta técnica lleva el adjetivo *naive*, ingenuo, en su nombre). Imaginemos que nuestro vocabulario solo consiste en las palabras "bitcoin" y Rolex, y que la mitad de los mensajes de *spam* son para "gana bitcoin" y la otra mitad son para "auténtico Rolex". En este caso, Naive Bayes estima que un mensaje de *spam* que contiene tanto "bitcoin" como "Rolex" es:

$$P(X_1 = 1, X_2 = 1|S) = P(X_1 = 1|S)P(X_2 = 1|S) = .5 \text{ x } .5 = .25$$

Ya que hemos asumido el saber que "bitcoin" y "Rolex" nunca ocurren realmente juntas. A pesar de la falta de realismo de esta suposición, este modelo suele funcionar bien y se ha utilizado desde siempre en filtros de *spam* reales.

El mismo razonamiento del teorema de Bayes que hemos utilizado para nuestro filtro de *spam* "solo bitcoin" nos dice que podemos calcular la probabilidad de que un mensaje sea *spam* utilizando la ecuación:

$$P(S|X = x) = P(X = x|S)/[P(X = x|S) + P(X = x|\neg S)]$$

La suposición de Naive Bayes nos permite calcular cada una de las probabilidades de la derecha simplemente multiplicando las estimaciones de probabilidad individuales por cada palabra del vocabulario.

En la práctica, nos interesará evitar multiplicar muchas probabilidades para no tener un problema llamado *underflow* (o desbordamiento por defecto), en el que los ordenadores no saben manejar bien números de punto flotante que son demasiado próximos a 0. Recordando de álgebra que log (ab) = log a + log b y que exp (log x) = x, normalmente calculamos p_1 * ... * p_n como el equivalente (pero que maneja mejor el punto flotante):

$$\exp (\log (p_1) + ... + \log (p_n))$$

El único desafío que nos queda viene con las estimaciones para $P(X_i|S)$ y $P(X_i|\neg S)$, las probabilidades de que un mensaje que es *spam* (o que no es *spam*) contenga la palabra "w_i". Si tenemos un buen número de mensajes de

“entrenamiento” etiquetados como *spam* y no *spam*, un obvio primer intento es estimar $P(X_i|S)$ simplemente como la fracción de mensajes de *spam* que contienen la palabra “w_i”.

Pero esto provoca un gran problema. Supongamos que en nuestro conjunto de entrenamiento la palabra del vocabulario datos solo aparece en mensajes que no son *spam*. Entonces estimaríamos $P(\text{datos}|S) = 0$. El resultado es que nuestro clasificador Naive Bayes siempre asignaría probabilidad de *spam* 0 a cualquier mensaje que contuviera la palabra “datos”, incluso a un mensaje como “datos sobre bitcoin gratis y auténticos relojes Rolex”. Para evitar este problema, normalmente empleamos algún tipo de suavizado.

En particular, elegiremos un pseudocontador (k) y estimaremos la probabilidad de ver la palabra i-ésima en un mensaje de *spam* como:

$P(X_i|S)$ = (k + número de spams que contienen w_i)/(2k + número de spams)

Hacemos lo mismo para $P(X_i|\neg S)$. Es decir, cuando calculemos las probabilidades de *spam* para la palabra i-ésima, supondremos que también vimos k mensajes adicionales que no son *spam* y que contienen la palabra y k mensajes adicionales que no son *spam* y que no contienen la palabra.

Por ejemplo, si datos aparece en 0/98 mensajes de *spam*, y si k es 1, estimamos $P(\text{datos}|S)$ como 1/100 = 0,01, lo que permite a nuestro clasificador seguir asignando una cierta probabilidad de *spam* no cero a mensajes que contienen la palabra “datos”.

Implementación

Ahora que tenemos todas las piezas, debemos montar nuestro clasificador. Primero, creamos una sencilla función con `tokenize` para separar los mensajes en palabras. Convertiremos antes cada mensaje a minúsculas, y emplearemos después `re.findall` para extraer “palabras” formadas por

letras, números y apóstrofos. Para terminar, utilizamos `set` para obtener las palabras por separado.

```
from typing import Set
import re
def tokenize(text: str) -> Set[str]:
   text = text.lower()                            # Convierte a minúsculas,
   all_words = re.findall("[a-z0-9’]+", text)     # extrae las palabras y
   return set(all_words)                          # elimina duplicados.
assert tokenize("Data Science is science") == {"data", "science", "is"}
```

También definiremos un tipo para nuestros datos de entrenamiento:

```
from typing import NamedTuple
class Message(NamedTuple):
   text: str
   is_spam: bool
```

Como nuestro clasificador tiene que controlar los *tokens*, contadores y etiquetas en los datos de entrenamiento, le crearemos una clase. Siguiendo el convenio, denominaremos emails “*ham*” a los mensajes que no son *spam*.

El constructor tomará solo un parámetro, el pseudocontador que se utiliza al calcular las probabilidades. También inicializa un conjunto vacío de *tokens*, contadores para controlar la frecuencia con la que cada *token* es visto en mensajes de *spam* y *ham*, y contadores de la cantidad de mensajes de *spam* y *ham* en los que fue entrenado:

```
from typing import List, Tuple, Dict, Iterable
import math
from collections import defaultdict
class NaiveBayesClassifier:
   def __init__(self, k: float = 0.5) -> None:
      self.k = k              # factor suavizante
      self.tokens: Set[str] = set()
      self.token_spam_counts: Dict[str, int] = defaultdict(int)
      self.token_ham_counts: Dict[str, int] = defaultdict(int)
      self.spam_messages = self.ham_messages = 0
```

A continuación, le daremos un método para entrenarlo con un montón de

mensajes. Primero, aumentamos los contadores de `spam_messages` y `ham_messages`. Después, dividimos cada mensaje de texto con `tokenize` y por cada *token* incrementamos los `token_spam_counts` o `token_ham_counts` según el tipo de mensaje:

```
def train(self, messages: Iterable[Message]) -> None:
    for message in messages:
        # Incrementa contadores de mensajes
        if message.is_spam:
            self.spam_messages += 1
        else:
            self.ham_messages += 1
        # Incrementa contadores de palabras
        for token in tokenize(message.text):
            self.tokens.add(token)
            if message.is_spam:
                self.token_spam_counts[token] += 1
            else:
                self.token_ham_counts[token] += 1
```

Lo que queremos en última instancia es predecir *P*(spam | token). Como ya hemos visto antes, para aplicar el teorema de Bayes tenemos que conocer *P*(token | spam) y *P*(token | ham) para cada *token* del vocabulario. De modo que crearemos una función auxiliar “privada” para calcularlos:

```
def _probabilities(self, token: str) -> Tuple[float, float]:
    """returns P(token | spam) and P(token | ham)"""
    spam = self.token_spam_counts[token]
    ham = self.token_ham_counts[token]
    p_token_spam = (spam + self.k) / (self.spam_messages + 2 * self.k)
    p_token_ham = (ham + self.k) / (self.ham_messages + 2 * self.k)
    return p_token_spam, p_token_ham
```

Finalmente, estamos listos para escribir nuestro modelo `predict`. Como dijimos antes, en lugar de multiplicar muchas probabilidades pequeñas, sumaremos las probabilidades logarítmicas:

```
def predict(self, text: str) -> float:
    text_tokens = tokenize(text)
    log_prob_if_spam = log_prob_if_ham = 0.0
```

```
        # Itera por cada palabra de nuestro vocabulario
        for token in self.tokens:
            prob_if_spam, prob_if_ham = self._probabilities(token)
            # Si aparece *token* en el mensaje,
            # añade la probabilidad logarítmica de verlo
            if token in text_tokens:
                log_prob_if_spam += math.log(prob_if_spam)
                log_prob_if_ham += math.log(prob_if_ham)
            # En otro caso añade la probabilidad logarítmica de _no_ verlo,
            # que es log(1 – probabilidad de verlo)
            else:
                log_prob_if_spam += math.log(1.0–prob_if_spam)
                log_prob_if_ham += math.log(1.0–prob_if_ham)
        prob_if_spam = math.exp(log_prob_if_spam)
        prob_if_ham = math.exp(log_prob_if_ham)
        return prob_if_spam / (prob_if_spam + prob_if_ham)
```

Y ya tenemos un clasificador.

A probar nuestro modelo

Asegurémonos de que nuestro modelo funciona escribiendo para él unas unidades de prueba.

```
messages = [Message("spam rules", is_spam=True),
            Message("ham rules", is_spam=False),
            Message("hello ham", is_spam=False)]
model = NaiveBayesClassifier(k=0.5)
model.train(messages)
```

Primero, veamos si obtuvo correctamente los contadores:

```
assert model.tokens == {"spam", "ham", "rules", "hello"}
assert model.spam_messages == 1
assert model.ham_messages == 2
assert model.token_spam_counts == {"spam": 1, "rules": 1}
assert model.token_ham_counts == {"ham": 2, "rules": 1, "hello": 1}
```

Ahora hagamos una predicción. También revisaremos (laboriosamente) a mano nuestra lógica Naive Bayes, y nos aseguraremos de que obtenemos el mismo resultado:

```
text = "hello spam"
probs_if_spam = [
   (1 + 0.5) / (1 + 2 * 0.5), # "spam" (presente)
   1-(0 + 0.5) / (1 + 2 * 0.5), # "ham" (no presente)
   1-(1 + 0.5) / (1 + 2 * 0.5), # "rules" (no presente)
   (0 + 0.5) / (1 + 2 * 0.5) # "hello" (presente)
]
probs_if_ham = [
   (0 + 0.5) / (2 + 2 * 0.5), # "spam" (presente)
   1-(2 + 0.5) / (2 + 2 * 0.5), # "ham" (no presente)
   1-(1 + 0.5) / (2 + 2 * 0.5), # "rules" (no presente)
   (1 + 0.5) / (2 + 2 * 0.5), # "hello" (presente)
]
p_if_spam = math.exp(sum(math.log(p) for p in probs_if_spam))
p_if_ham = math.exp(sum(math.log(p) for p in probs_if_ham))
# Debería ser como 0,83
assert model.predict(text) == p_if_spam / (p_if_spam + p_if_ham)
```

Esta prueba funciona, de modo que parece que nuestro modelo está haciendo lo que pensamos que debe hacer. Si miramos las probabilidades reales, los dos grandes conductores son que nuestro mensaje contiene *spam* (cosa que hacía nuestro único mensaje *spam* de entrenamiento) y que no contiene *ham* (cosa que hacían nuestros dos mensajes *ham* de entrenamiento).

A continuación, probemos con datos de verdad.

Utilizar nuestro modelo

Un conjunto de datos conocido (aunque algo antiguo) es el recopilatorio público SpamAssassin.[1] Vamos a consultar los archivos con el prefijo 20021010.

Este es un fragmento de código que los descargará y descomprimirá en el directorio que elijamos (o bien puede hacerse manualmente):

```
from io import BytesIO      # Así podemos tratar bytes como archivo.
import requests             # Descargar los archivos, que
import tarfile              # están en formato .tar.bz.
BASE_URL = "https://spamassassin.apache.org/old/publiccorpus"
FILES = ["20021010_easy_ham.tar.bz2",
     "20021010_hard_ham.tar.bz2",
```

```
    "20021010_spam.tar.bz2"]
# Aquí terminarán los datos,
# en los subdirectorios /spam, /easy_ham y /hard_ham.
# Cambie esto a donde quiera los datos.
OUTPUT_DIR = 'spam_data'
for filename in FILES:
  # Usa requests para obtener el contenido del archivo en cada URL.
  content = requests.get(f"{BASE_URL}/{filename}").content
  # Envuelve los bytes en memoria para poder usarlos como "archivo".
  fin = BytesIO(content)
  # Y extrae todos los archivos al dir de salida especificado.
  with tarfile.open(fileobj=fin, mode='r:bz2') as tf:
    tf.extractall(OUTPUT_DIR)
```

Es posible que la ubicación de los archivos cambie (lo que ocurrió entre la primera y segunda edición de este libro), en cuyo caso ajuste el código adecuadamente.

Tras descargar los datos, debería tener tres carpetas: `spam`, `easy_ham` y `hard_ham`. Cada carpeta contiene muchos emails, cada uno de ellos contenido en un solo archivo. Para simplificar de verdad las cosas, solo veremos las líneas del asunto de cada email.

¿Cómo identificamos la línea del asunto? Cuando revisamos los archivos, todos parecen empezar por "Subject:" (asunto en inglés). Así que buscaremos eso:

```
import glob, re
# modifique la ruta a donde tenga los archivos
path = 'spam_data/*/*'
data: List[Message] = []
# glob.glob devuelve nombres de archivo que coinciden con la ruta comodín
for filename in glob.glob(path):
  is_spam = "ham" not in filename
  # Hay caracteres sobrantes en los emails; el errors='ignore'
  # los salta en lugar de mostrar una excepción.
  with open(filename, errors='ignore') as email_file:
    for line in email_file:
      if line.startswith("Subject:"):
        subject = line.lstrip("Subject: ")
        data.append(Message(subject, is_spam))
        break                # done with this file
```

Ahora podemos dividir los datos en datos de entrenamiento y datos de prueba, así estamos listos para crear un clasificador:

```
import random
from scratch.machine_learning import split_data
random.seed(0)      # justo así obtiene las mismas respuestas que yo
train_messages, test_messages = split_data(data, 0.75)
model = NaiveBayesClassifier()
model.train(train_messages)
```

Generemos algunas predicciones y veamos si funciona nuestro modelo:

```
from collections import Counter
predictions = [(message, model.predict(message.text))
           for message in test_messages]
# Supone que spam_probability > 0.5 corresponde a predicción de spam
# y cuenta las combinaciones de (actual is_spam, predicted is_spam)
confusion_matrix = Counter((message.is_spam, spam_probability > 0.5)
 for message, spam_probability in predictions)
print(confusion_matrix)
```

Esto proporciona 84 verdaderos positivos (*spam* clasificado como "*spam*"), 25 falsos positivos (han clasificado como "*spam*"), 703 verdaderos negativos (*ham* clasificado como "*ham*") y 44 falsos negativos (*spam* clasificado como "*ham*"), lo que significa que nuestra precisión es de 84 / (84 + 25) = 77 % y nuestro recuerdo de 84 / (84 + 44) = 65 %, que no son números malos para un modelo tan sencillo (supuestamente saldría mejor si tuviéramos en cuenta algo más que las líneas del asunto).

También podemos inspeccionar las entrañas del modelo para ver qué palabras son menos y más indicadoras de *spam*:

```
def p_spam_given_token(token: str, model: NaiveBayesClassifier) -> float:
   # Probablemente no habría que usar métodos privados, pero es por una buena
   causa.
   prob_if_spam, prob_if_ham = model._probabilities(token)
   return prob_if_spam / (prob_if_spam + prob_if_ham)
words = sorted(model.tokens, key=lambda t: p_spam_given_token(t, model))
print("spammiest_words", words[-10:])
print("hammiest_words", words[:10])
```

Entre las palabras más indicadoras de *spam* se encuentran *sale* (venta), *mortgage* (hipoteca), *money* (dinero) y *rates* (cuotas), mientras que algunas de las palabras más indicadoras de *ham* son *spambayes*, *users* (usuarios), *apt* y *perl*. Por lo tanto, esto nos da cierta confianza intuitiva en que nuestro modelo está haciendo básicamente lo que debe hacer.

¿Cómo podemos obtener un mejor rendimiento? Una forma obvia podría ser consiguiendo más datos para entrenar el modelo. También hay otras maneras diferentes de mejorar el modelo; estas son algunas posibilidades:

- Mirar el contenido del mensaje, no solo el asunto. Hay que tener cuidado con el manejo de los encabezados de los mensajes.
- Nuestro clasificador tiene en cuenta cada palabra que aparece en el conjunto de entrenamiento, incluso palabras que solo aparecen una vez. Se puede modificar para que acepte un umbral óptimo `min_count` e ignore *tokens* que no aparecen al menos esa cantidad de veces.
- El `tokenizer` no tiene la noción de palabras similares (por ejemplo, *cheap* y *cheapest*). Entonces se puede cambiar el clasificador para que admita una función opcional `stemmer`, que convierte palabras en clases de equivalencia de palabras. Por ejemplo, una función stemmer muy sencilla podría ser:

```
def drop_final_s(word):
  return re.sub("s$", "", word)
```

Crear una buena función stemmer es difícil. La gente suele utilizar el stemmer Porter.[2]

- Aunque nuestras funciones tengan la forma “el mensaje contiene la palabra w_i”, no hay razón por la que este tenga que ser el caso. En nuestra implementación, podríamos añadir funciones adicionales como “el mensaje contiene un número” creando *tokens* falsos como *contains:number* y modificando el `tokenizer` para emitirlos cuando corresponda.

Para saber más

- Los artículos de Paul Graham, “A Plan for Spam” en `http://www.paulgraham.com/spam.html` y “Better Bayesian Filtering” en (`http://www.paulgraham.com/better.html`, son interesantes y dan más información sobre las ideas que subyacen tras la creación de filtros de *spam*.
- scikit-learn, en `https://scikit-learn.org/stable/modules/naive_bayes.html`, contiene un modelo `BernoulliNB` que implementa el mismo algoritmo Naive Bayes que hemos implementado aquí, además de otras variaciones del modelo.

---

[1] `https://spamassassin.apache.org/old/publiccorpus/`.

[2] `http://tartarus.org/martin/PorterStemmer/`.

14 Regresión lineal simple

El arte, como la moral, consiste en trazar la línea en algún lugar.

—G. K. Chesterton

En el capítulo 5 utilizamos la función `correlation` para medir la fuerza de la relación lineal entre dos variables. En la mayoría de las aplicaciones, no basta con saber que existe una relación lineal como esta; queremos comprender la naturaleza de la relación. Aquí es donde empleamos la regresión lineal simple.

El modelo

Recordemos que estábamos investigando la relación entre el número de amigos de un usuario de DataSciencester y la cantidad de tiempo que el usuario se pasa en el sitio cada día. Supongamos que usted se ha convencido a sí mismo de que tener más amigos hace que la gente se pase más tiempo en el sitio, en lugar de una de las explicaciones alternativas que ya hemos tratado.

El vicepresidente de Compromiso le pide que cree un modelo que describa esta relación. Como ha descubierto ya una relación lineal bastante intensa, lo más natural es empezar por un modelo lineal.

En particular, podríamos proponer que haya dos constantes α (alfa) y β (beta) como por ejemplo:

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

Donde *yi* es el número de minutos que el usuario *i* se pasa en el sitio diariamente, *xi* es el número de amigos que tiene el usuario *i*, y ε es un término de error (con suerte, pequeño) que representa el hecho de que hay

otros factores no tenidos en cuenta en este sencillo modelo.

Suponiendo que hemos determinado tales `alpha` y `beta`, hacemos predicciones fácilmente con:

```
def predict(alpha: float, beta: float, x_i: float) -> float:
   return beta * x_i + alpha
```

¿Cómo elegimos `alpha` y `beta`? Pues sea cual sea los que elijamos, nos dan un resultado previsto para cada entrada `x_i`. Como ya conocemos el resultado real `y_i`, podemos calcular el error para cada par:

```
def error(alpha: float, beta: float, x_i: float, y_i: float) -> float:
   """
   The error from predicting beta * x_i + alpha
   when the actual value is y_i
   """
   return predict(alpha, beta, x_i)–y_i
```

Lo que realmente nos gustaría saber es el error total sobre el conjunto de datos completo. Pero no queremos solamente sumar los errores (si la predicción para `x_1` es demasiado alta y para `x_2` es demasiado baja, los errores podrían anularse mutuamente).

Así que, en lugar de ello, sumamos los errores al cuadrado:

```
from scratch.linear_algebra import Vector
def sum_of_sqerrors(alpha: float, beta: float, x: Vector, y: Vector) -> float:
   return sum(error(alpha, beta, x_i, y_i) ** 2
           for x_i, y_i in zip(x, y))
```

La solución de mínimos cuadrados es elegir los `alpha` y `beta` que hagan que `sum_of_sqerrors` sea lo más pequeña posible.

Utilizando el cálculo (o la aburrida álgebra), los `alpha` y `beta` minimizadores de errores vienen dados por:

```
from typing import Tuple
from scratch.linear_algebra import Vector
from scratch.statistics import correlation, standard_deviation, mean
```

```
def least_squares_fit(x: Vector, y: Vector) -> Tuple[float, float]:
   """
   Given two vectors x and y,
   find the least-squares values of alpha and beta
   """
   beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
   alpha = mean(y)–beta * mean(x)
   return alpha, beta
```

Sin revisar a fondo las matemáticas exactas, pensemos en la razón por la que esta podría ser una solución razonable. La elección de `alpha` dice simplemente que, cuando vemos el valor medio de la variable independiente `x`, predecimos el valor medio de la variable dependiente `y`.

La elección de `beta` significa que, cuando el valor de entrada aumenta en `standard_deviation(x)`, la predicción se incrementa entonces en `correlation(x, y) * standard_deviation(y)`. En el caso en que `x` e `y` estén perfectamente correlacionadas, un incremento de una sola desviación estándar en `x` da como resultado un aumento de una sola desviación estándar de `y` en la predicción. Cuando están perfectamente anticorrelacionadas, el aumento en `x` da como resultado una disminución en la predicción. Y, cuando la correlación es 0, `beta` es 0, lo que significa que los cambios en `x` no afectan en absoluto a la predicción.

Como es habitual, escribimos una rápida prueba para esto:

```
x = [i for i in range(-100, 110, 10)]
y = [3 * i–5 for i in x]
# Debería hallar que y = 3x–5
assert least_squares_fit(x, y) == (-5, 3)
```

Ahora es fácil aplicar esto a los datos sin *outliers* del capítulo 5:

```
from scratch.statistics import num_friends_good, daily_minutes_good
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
assert 22.9 < alpha < 23.0
assert 0.9 < beta < 0.905
```

Lo que nos da valores de `alpha` = 22,95 y `beta` = 0,903. Por tanto, nuestro

modelo dice que esperamos que un usuario con *n* amigos pase 22,95 + *n* * 0,903 minutos en el sitio cada día. Es decir, predecimos que un usuario sin amigos en DataSciencester aún se pasaría unos 23 minutos al día en el sitio. Por cada amigo adicional, esperamos que un usuario se pase casi un minuto más en el sitio cada día.

En la figura 14.1 trazamos la línea de predicción para hacernos una idea de lo bien que encaja el modelo en los datos observados.

Por supuesto, necesitamos una manera mejor de averiguar lo bien que hemos ajustado los datos que simplemente mirando el gráfico. Un método habitual es el coeficiente de determinación (o R^2, pronunciado R cuadrado), que mide la fracción de la variación total en la variable dependiente que es capturada por el modelo:

```
from scratch.statistics import de_mean
def total_sum_of_squares(y: Vector) -> float:
   """the total squared variation of y_i's from their mean"""
   return sum(v ** 2 for v in de_mean(y))
def r_squared(alpha: float, beta: float, x: Vector, y: Vector) -> float:
   """
   the fraction of variation in y captured by the model, which equals
   1-the fraction of variation in y not captured by the model
   """
   return 1.0-(sum_of_sqerrors(alpha, beta, x, y) /
           total_sum_of_squares(y))
rsq = r_squared(alpha, beta, num_friends_good, daily_minutes_good)
assert 0.328 < rsq < 0.330
```

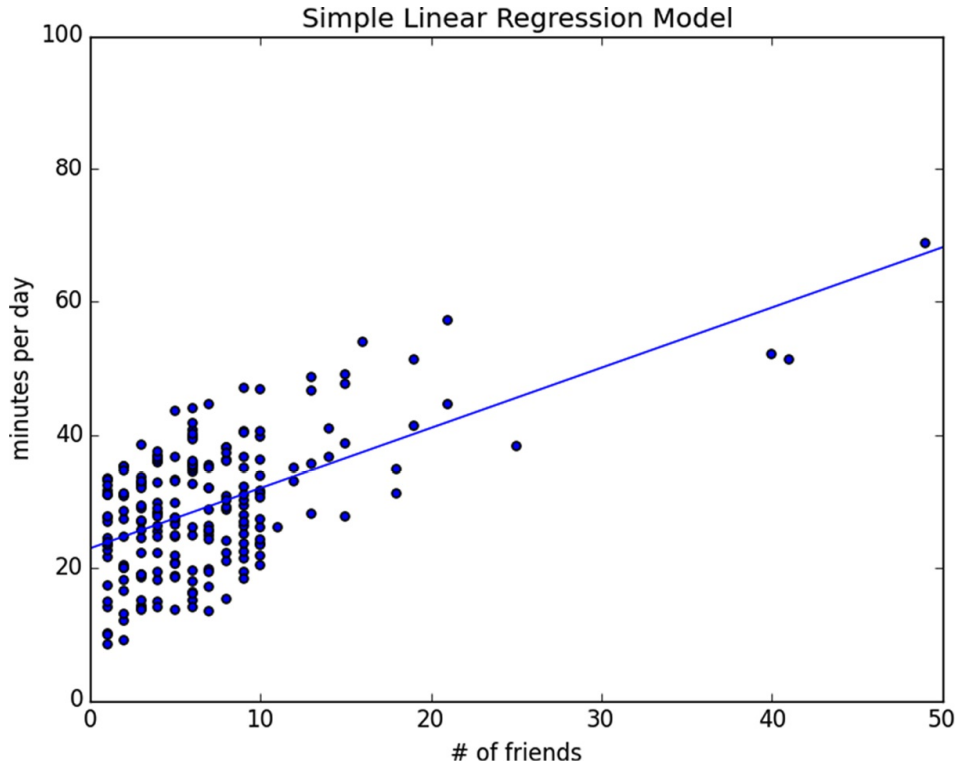


Figura 14.1. Nuestro sencillo modelo lineal.

Recordemos que elegimos los `alpha` y `beta` que minizaban la suma de los errores de predicción al cuadrado. Un modelo lineal que pudimos haber elegido es "predecir siempre `mean(y)`" (correspondiendo a `alpha` = `mean(y)` y `beta` = 0), cuya suma de errores al cuadrado es exactamente igual a su suma total de cuadrados. Esto significa un R cuadrado de 0, que indica un modelo que (obviamente, en este caso) no funciona mejor que solamente predecir la media.

Sin duda, el modelo de mínimos cuadrados debe ser al menos tan bueno como este, lo que significa que la suma de los errores al cuadrado es como mucho la suma total de cuadrados, es decir, que el R cuadrado debe ser al menos 0. Y la suma de errores al cuadrado debe ser al menos 0, lo que significa que el R cuadrado puede ser como mucho 1.

Cuánto más alto sea el número, mejor ajustará nuestro modelo los datos. Aquí calculamos un R cuadrado de 0,329, lo que nos dice que nuestro modelo solo se ajusta más o menos bien a los datos, y que sin duda hay otros factores en juego.

Utilizar descenso de gradiente

Si escribimos `theta = [alpha, beta]`, podemos también resolver esto utilizando descenso de gradiente:

```
import random
import tqdm
from scratch.gradient_descent import gradient_step
num_epochs = 10000
random.seed(0)
guess = [random.random(),                    # selecciona valor aleatorio para
random.random()]                             empezar
learning_rate = 0.00001
with tqdm.trange(num_epochs) as t:
   for _ in t:
      alpha, beta = guess
      # Derivada parcial de pérdida con respecto a alpha
      grad_a = sum(2 * error(alpha, beta, x_i, y_i)
                for x_i, y_i in zip(num_friends_good,
                    daily_minutes_good))
      # Derivada parcial de pérdida con respecto a beta
      grad_b = sum(2 * error(alpha, beta, x_i, y_i) * x_i
                for x_i, y_i in zip(num_friends_good,
                    daily_minutes_good))
      # Calcula pérdida para mantener la descripción tqdm
      loss = sum_of_sqerrors(alpha, beta,
num_friends_good, daily_minutes_good)
      t.set_description(f"loss: {loss:.3f}")
      # Por último, actualiza la conjetura
      guess = gradient_step(guess, [grad_a, grad_b], -learning_rate)
# Deberíamos obtener resultados similares:
alpha, beta = guess
assert 22.9 < alpha < 23.0
assert 0.9 < beta < 0.905
```

Si ejecutamos esto obtendremos los mismos valores para `alpha` y `beta` que obtuvimos utilizando la fórmula exacta.

Estimación por máxima verosimilitud

¿Por qué elegir mínimos cuadrados? Una justificación tiene que ver con la estimación por máxima verosimilitud. Supongamos que tenemos una muestra de datos *v*1, ..., *vn* procedente de una distribución que depende de un cierto parámetro desconocido θ (theta):

$$p(v_1, ..., v_n \mid \theta)$$

Si no conocemos θ, podríamos sentarnos a pensar en esta cantidad como en la verosimilitud de θ dada la muestra:

$$L(\theta \mid v_1, ..., v_n)$$

Según este enfoque, el parámetro θ más admisible es el valor que maximiza esta función de verosimilitud (es decir, el valor que hace que el dato observado sea el más probable). En el caso de una distribución continua, en la que tenemos una función de distribución de probabilidad en lugar de una función de masa de probabilidad, podemos hacer lo mismo.

Volvamos a la regresión. Una suposición que se hace a menudo sobre el modelo de regresión simple es que los errores de regresión se distribuyen normalmente con media 0 y una cierta desviación estándar (conocida) σ. Si es este el caso, entonces la verosimilitud basada en ver un par (`x_i`, `y_i`) es:

$$L(\alpha, \beta \mid x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-(y_i - \alpha - \beta x_i)^2 / 2\sigma^2\right)$$

La verosimilitud basada en el conjunto de datos completo es el producto de las verosimilitudes individuales, que es mayor precisamente cuando `alpha` y `beta` se eligen para minimizar la suma de errores al cuadrado. Es decir, en este caso (con estas suposiciones), minimizar la suma de errores al cuadrado es equivalente a maximizar la verosimilitud de los datos observados.

Para saber más

Siga leyendo sobre la regresión múltiple en el capítulo 15.

15 Regresión múltiple

Yo no miro un problema y pongo en él variables que no le afectan.

—Bill Parcells

Aunque la vicepresidenta está bastante impresionada con su modelo predictivo, ella cree que puede hacerlo mejor. Con ese objetivo, se ha dedicado a recoger datos adicionales: sabe cuántas horas trabaja cada día cada uno de sus usuarios y si tienen un doctorado; naturalmente, quiere utilizar estos datos para mejorar su modelo.

Por consiguiente, se le ocurre proponer un modelo lineal con más variables independientes:

$$\text{minutos} = \alpha + \beta_1 \text{amigos} + \beta_2 \text{horas de trabajo} + \beta_3 \text{doctorado} + \varepsilon$$

Obviamente, el hecho de que un usuario tenga un doctorado no es un número, pero, como ya vimos en el capítulo 11, podemos introducir una variable ficticia que es igual a 1 para usuarios con doctorado y a 0 para los que no lo tienen, y así es igual de numérica que las demás variables.

El modelo

Recordemos que en el capítulo 14 ajustamos un modelo con esta forma:

$$y_i \; \alpha + \beta x_i \; \varepsilon_i$$

Ahora supongamos que cada entrada x_i no es solamente un número, sino más bien un vector de k números x_{i1}, ..., x_{ik}. El modelo de regresión múltiple supone que:

$$y_i = \alpha \; \beta_1 x i_1 \; \beta_k x_{ik} \; \varepsilon_i$$

En la regresión múltiple el vector de parámetros se suele denominar β. Queremos que incluya también el término constante, cosa que podemos conseguir añadiendo una columna de unos a nuestros datos:

```
beta = [alpha, beta_1, ..., beta_k]
```

Y:

```
x_i = [1, x_i1, ..., x_ik]
```

Entonces nuestro modelo queda así:

```
from scratch.linear_algebra import dot, Vector
def predict(x: Vector, beta: Vector) -> float:
   """assumes that the first element of x is 1"""
   return dot(x, beta)
```

En este caso en particular, nuestra variable independiente `x` será una lista de vectores, cada uno de los cuales tiene este aspecto:

```
[1,     # término constante
49,     # número de amigos
4,      # horas de trabajo al día
0]      # no tiene doctorado
```

Otros supuestos del modelo de mínimos cuadrados

Hay un par de supuestos adicionales necesarios para que este modelo (y nuestra solución) tenga sentido.

El primero es que las columnas de *x* son linealmente independientes (es decir, que no hay forma de escribir cualquiera de ellas como una suma ponderada de algunas de las otras). Si este supuesto falla, es imposible estimar `beta`. Para ver esto en una situación extrema, imaginemos que tuviéramos en nuestros datos un campo adicional `num_acquaintances` que, para cada usuario, fuera exactamente igual a `num_friends`.

Después, empezando con cualquier `beta`, si sumamos cualquier número al

coeficiente `num_friends` y restamos ese mismo número al coeficiente `num_acquaintances`, las predicciones del modelo no cambiarán, lo que significa que no hay forma de encontrar el coeficiente para `num_friends` (normalmente, los incumplimientos de este supuesto no serán tan obvios).

El segundo supuesto importante es que las columnas de *x* no están todas correlacionadas con los errores ε. Si este no es el caso, nuestras estimaciones de `beta` serán sistemáticamente erróneas. Por ejemplo, en el capítulo 14 creamos un modelo que predecía que cada amigo adicional estaba asociado a 0,90 minutos diarios extra en el sitio. Supongamos que es también el caso que:

- La gente que trabaja más horas se pasa menos tiempo en el sitio.
- La gente con más amigos tiende a trabajar más horas.

Es decir, imaginemos que el modelo "real" es:

$$\text{minutos} = \alpha + \beta_1 \text{amigos} + \beta_2 \text{horas de trabajo} + \varepsilon$$

Donde β_2 es negativo, y que las horas de trabajo y los amigos están positivamente correlacionados. En ese caso, cuando minimicemos los errores del modelo de única variable:

$$\text{minutos} = \alpha + \beta_1 \text{amigos} + \varepsilon$$

Subestimaremos β_1.

Pensemos en lo que ocurriría si hiciéramos predicciones utilizando el modelo de única variable con el valor "real" de $\beta 1$ (es decir, el valor que se obtiene de minimizar los errores de lo que llamamos el modelo "real"). Las predicciones tenderían a ser excesivamente grandes para usuarios que trabajan muchas horas y un poco demasiado grandes para los que trabajan pocas horas, porque $\beta 2 < 0$ y "olvidamos" incluirlo. El hecho de que las horas de trabajo estén positivamente correlacionadas con el número de amigos significa que las predicciones tienden a ser excesivamente grandes para los usuarios con muchos amigos, y solo ligeramente demasiado grandes para

usuarios con pocos amigos.

El resultado es que podemos reducir los errores (en el modelo de única variable) disminuyendo nuestra estimación de β_1, lo que significa que la β_1 minimizadora de errores es más pequeña que el valor "real". Es decir, en este caso la solución de mínimos cuadrados de única variable tiende a subestimar β_1. Además, en general, siempre que las variables independientes estén correlacionadas con errores como este, nuestra solución de mínimos cuadrados nos dará una estimación sesgada de β_1.

Ajustar el modelo

Como hicimos en el modelo lineal simple, elegiremos `beta` para minimizar la suma de errores cuadrados. Hallar una solución exacta manualmente no es sencillo, lo que significa que tendremos que utilizar descenso de gradiente. De nuevo, nos interesará minimizar la suma de los errores cuadrados. La función de error es casi idéntica a la que empleamos en el capítulo 14, salvo porque, en lugar de esperar parámetros `[alpha, beta]`, requerirá un vector de longitud arbitraria:

```
from typing import List
def error(x: Vector, y: float, beta: Vector) -> float:
   return predict(x, beta)–y
def squared_error(x: Vector, y: float, beta: Vector) -> float:
   return error(x, y, beta) ** 2
x = [1, 2, 3]
y = 30
beta = [4, 4, 4]       # así la predicción = 4 + 8 + 12 = 24
assert error(x, y, beta) == -6
assert squared_error(x, y, beta) == 36
```

Sabiendo cálculo, es fácil calcular el gradiente:

```
def sqerror_gradient(x: Vector, y: float, beta: Vector) -> Vector:
   err = error(x, y, beta)
   return [2 * err * x_i for x_i in x]
assert sqerror_gradient(x, y, beta) == [-12, -24, -36]
```

Si no, habrá que aceptar mi palabra.

En este momento, estamos preparados para encontrar la `beta` óptima utilizando descenso de gradiente. Escribamos primero una función `least_squares_fit` que pueda funcionar con cualquier conjunto de datos:

```
import random
import tqdm
from scratch.linear_algebra import vector_mean
from scratch.gradient_descent import gradient_step
def least_squares_fit(xs: List[Vector],
                      ys: List[float],
                      learning_rate: float = 0.001,
                      num_steps: int = 1000,
                      batch_size: int = 1) -> Vector:
  """
  Find the beta that minimizes the sum of squared errors
  assuming the model y = dot(x, beta).
  """
  # Empieza con una conjetura aleatoria
  guess = [random.random() for _ in xs[0]]
  for _ in tqdm.trange(num_steps, desc="least squares fit"):
     for start in range(0, len(xs), batch_size):
        batch_xs = xs[start:start+batch_size]
        batch_ys = ys[start:start+batch_size]
        gradient = vector_mean([sqerror_gradient(x, y, guess)
for x, y in zip(batch_xs, batch_ys)])
        guess = gradient_step(guess, gradient, -learning_rate)
  return guess
```

Entonces podemos aplicarla a nuestros datos:

```
from scratch.statistics import daily_minutes_good
from scratch.gradient_descent import gradient_step
random.seed(0)
# Usé prueba y error para elegir num_iters y step_size.
# Esto funcionará un rato.
learning_rate = 0.001
beta = least_squares_fit(inputs, daily_minutes_good, learning_rate, 5000, 25)
assert 30.50 < beta[0] < 30.70                # constante
assert 0.96 < beta[1] < 1.00                  # núm amigos
assert -1.89 < beta[2] < -1.85                # horas trabajo al día
assert 0.91 < beta[3] < 0.94                  # doctorado
```

En la práctica, no estimaríamos una regresión lineal utilizando descenso de gradiente; se obtendrían los coeficientes exactos empleando técnicas de álgebra lineal que están más allá del alcance de este libro. Si lo hiciéramos, daríamos con la ecuación:

minutos = 30 , 58 + 0 , 972 amigos – 1 , 87 horas de trabajo + 0 , 923 doctorados

Que se acerca bastante a lo que hemos descubierto.

Interpretar el modelo

Hay que pensar que los coeficientes del modelo representan estimaciones del tipo “siendo todo lo demás igual” de los impactos que tiene cada factor. Siendo todo lo demás igual, cada amigo adicional corresponde con un minuto extra pasado cada día en el sitio. Siendo todo lo demás igual, cada hora adicional del día de trabajo de un usuario corresponde con unos dos minutos menos pasados cada día en el sitio. Siendo todo lo demás igual, tener un doctorado se asocia a pasar un minuto más cada día en el sitio.

Lo que no nos dice (directamente) es nada sobre las interacciones entre las variables. Es posible que el efecto de las horas de trabajo sea diferente en gente con muchos amigos que en gente con pocos. Este modelo no captura eso. Una forma de manejar esta situación es introducir una nueva variable: el producto de “amigos” y “horas de trabajo”. Esto permite sin duda al coeficiente de “horas de trabajo” aumentar (o disminuir) a medida que el número de amigos se incrementa.

También es posible que cuantos más amigos se tengan, más tiempo se pase en el sitio hasta un cierto punto, tras el cual aún más amigos hace que se pase menos tiempo en el sitio (¿quizá con demasiados amigos la experiencia es demasiado abrumadora?). Podríamos intentar capturar esto en nuestro modelo añadiendo otra variable que es el cuadrado del número de amigos.

En cuanto empezamos a añadir variables, tenemos que preocuparnos de si sus coeficientes “importan”. No hay límites en las cantidades de productos,

logaritmos, cuadrados y potencias que podemos añadir.

Bondad de ajuste

De nuevo podemos echar un vistazo al R cuadrado:

```
from scratch.simple_linear_regression import total_sum_of_squares
def multiple_r_squared(xs: List[Vector], ys: Vector, beta: Vector) -> float:
   sum_of_squared_errors = sum(error(x, y, beta) ** 2
 for x, y in zip(xs, ys))
   return 1.0-sum_of_squared_errors / total_sum_of_squares(ys)
```

Que ha aumentado ahora a 0,68:

```
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta) < 0.68
```

Debemos recordar, sin embargo, que añadir nuevas variables a una regresión aumentará necesariamente el R cuadrado. Después de todo, el modelo de regresión simple no es más que el caso especial del modelo de regresión múltiple, en el que los coeficientes de "horas de trabajo" y "doctorado" son ambos iguales a 0. El modelo de regresión múltiple óptimo tendrá obligadamente un error al menos tan pequeño como ese.

Debido a esto, en una regresión múltiple también tenemos que mirar los errores estándares de los coeficientes, que miden lo seguros que estamos de nuestras estimaciones de cada β_1. La regresión, como un todo, puede ajustar muy bien nuestros datos, pero, si algunas de las variables independientes están correlacionadas (o son irrelevantes), sus coeficientes podrían no significar mucho.

El enfoque habitual para medir estos errores empieza con otro supuesto: que los errores ε_i son variables aleatorias normales e independientes con media 0 y una cierta desviación estándar (desconocida) σ. En ese caso, nosotros (o, lo más probable, nuestro software estadístico) podemos utilizar álgebra lineal para encontrar el error estándar de cada coeficiente. Cuanto más grande sea, menos seguro está nuestro modelo de ese coeficiente. Lamentablemente, no estamos preparados para este tipo de álgebra lineal

desde cero.

Digresión: el bootstrap

Supongamos que tenemos una muestra de *n* puntos de datos, generados por una cierta distribución (desconocida para nosotros):

```
data = get_sample(num_points=n)
```

En el capítulo 5, escribimos una función que podía calcular la `median` de la muestra, que podemos utilizar como estimación de la mediana de la propia distribución.

Pero ¿hasta qué punto podemos estar seguros de nuestra estimación? Si todos los puntos de datos de la muestra están muy cerca de 100, entonces parece probable que la mediana real esté cerca de 100. Si aproximadamente la mitad de los puntos de datos de la muestra está cerca de 0 y la otra mitad está cerca de 200, entonces no podemos estar tan seguros de la mediana.

Si pudiéramos obtener repetidamente nuevas muestras, podríamos calcular las medianas de todas esas muestras y mirar su distribución. Pero con frecuencia no es posible. En ese caso, sí se puede aplicar *bootstrap* a nuevos conjuntos de datos seleccionando *n* puntos de datos con reemplazo de nuestros datos. Después, podemos calcular las medianas de esos conjuntos de datos sintéticos:

```
from typing import TypeVar, Callable
X = TypeVar('X')               # Tipo genérico para datos
Stat = TypeVar('Stat')         # Tipo genérico para "estadística"
def bootstrap_sample(data: List[X]) -> List[X]:
   """randomly samples len(data) elements with replacement"""
   return [random.choice(data) for _ in data]
def bootstrap_statistic(data: List[X],
                  stats_fn: Callable[[List[X]], Stat],
                  num_samples: int) -> List[Stat]:
   """evaluates stats_fn on num_samples bootstrap samples from data"""
   return [stats_fn(bootstrap_sample(data)) for _ in range(num_samples)]
```

Por ejemplo, consideremos los dos conjuntos de datos siguientes:

```
# 101 puntos todos muy cerca de 100
close_to_100 = [99.5 + random.random() for _ in range(101)]
# 101 puntos, 50 de ellos cerca de 0, 50 de ellos cerca de 200
far_from_100 = ([99.5 + random.random()] +
           [random.random() for _ in range(50)] +
           [200 + random.random() for _ in range(50)])
```

Si calculamos las `median` de los dos conjuntos de datos, ambas estarán muy cerca de 100. Sin embargo, si miramos:

```
from scratch.statistics import median, standard_deviation
medians_close = bootstrap_statistic(close_to_100, median, 100)
```

Veremos en su mayoría números realmente cercanos a 100. Pero si lo que miramos es:

```
medians_far = bootstrap_statistic(far_from_100, median, 100)
```

Veremos muchos números cercanos a 0 y otros muchos cercanos a 200.

La `standard_deviation` del primer conjunto de medianas está cerca de 0, mientras que la del segundo conjunto de medianas está cerca de 100:

```
assert standard_deviation(medians_close) < 1
assert standard_deviation(medians_far) > 90
```

(Este caso extremo sería bastante fácil de averiguar manualmente inspeccionando los datos, pero en general eso no suele ocurrir).

Errores estándares de coeficientes de regresión

Podemos seguir el mismo sistema para estimar los errores estándares de nuestros coeficientes de regresión. Tomamos repetidamente una `bootstrap_sample` de nuestros datos y estimamos la `beta` basándonos en esa muestra. Si el coeficiente correspondiente a una de las variables independientes (digamos, `num_friends`) no varía mucho a lo largo de las muestras, entonces podemos estar seguros de que nuestra estimación es

relativamente ajustada. Si el coeficiente varía mucho a lo largo de las muestras, no podemos estar tan seguros entonces de nuestra estimación.

La única sutileza es que, antes de tomar las muestras, tendremos que empaquetar con `zip` nuestros datos `x` e `y` para asegurarnos de que los valores correspondientes de las variables independientes y dependientes estén juntos en la muestra. Esto significa que `bootstrap_sample` devolverá una lista de pares (`x_i`, `y_i`), que tendremos que volver a reformular como `x_sample` e `y_sample`:

```
from typing import Tuple
import datetime
def estimate_sample_beta(pairs: List[Tuple[Vector, float]]):
   x_sample = [x for x, _ in pairs]
   y_sample = [y for _, y in pairs]
   beta = least_squares_fit(x_sample, y_sample, learning_rate, 5000, 25)
   print("bootstrap sample", beta)
   return beta
random.seed(0)          # así obtiene los mismos datos que yo
# ¡Esto tardará un par de minutos!
bootstrap_betas = bootstrap_statistic(list(zip(inputs, daily_minutes_good)),
                   estimate_sample_beta,
                   100)
```

Después, podemos estimar la desviación estándar de cada coeficiente:

```
bootstrap_standard_errors = [
   standard_deviation([beta[i] for beta in bootstrap_betas])
   for i in range(4)]
print(bootstrap_standard_errors)
# [1.272,   # término constante,     error real = 1.19
# 0.103,    # num_friends,           error real = 0.080
# 0.155,    # work_hours,            error real = 0.127
# 1.249]    # doctorado,             error real = 0.998
```

(Obtendríamos mejores estimaciones si recogiéramos más de 100 muestras y utilizáramos más de 5.000 iteraciones para estimar cada `beta`, pero no tenemos todo el día).

Podemos utilizar estas estimaciones para probar hipótesis como "¿es β_i igual a 0?". Bajo la hipótesis nula $\beta_i = 0$ (y con nuestros otros supuestos

sobre la distribución de ε_i), la estadística:

$$t_j = \widehat{\beta_j}/\widehat{\sigma_j}$$

Que es nuestra estimación de β_j dividida por nuestra estimación de su error estándar, sigue a una distribución *t* de Student con "*n* – *k* grados de libertad".

Si tuviéramos una función `students_t_cdf`, podríamos calcular valores *p* para cada coeficiente de mínimos cuadrados, que indicara la probabilidad de que observáramos un valor así si el coeficiente real fuera 0. Lamentablemente, no tenemos una función como esta (aunque si la tuviéramos no estaríamos trabajando desde cero).

No obstante, a medida que los grados de libertad son mayores, la distribución *t* se acerca cada vez más a una normal estándar. En una situación como esta, donde *n* es mucho más grande que *k*, podemos utilizar `normal_cdf` y seguir sintiéndonos bien con nosotros mismos:

```
from scratch.probability import normal_cdf
def p_value(beta_hat_j: float, sigma_hat_j: float) -> float:
   if beta_hat_j > 0:
      # si el coeficiente es positivo, tenemos que calcular el doble
      # de la probabilidad de ver un valor aún *mayor*
      return 2 * (1–normal_cdf(beta_hat_j / sigma_hat_j))
   else:
      # si no el doble de la probabilidad de ver un valor *menor*
      return 2 * normal_cdf(beta_hat_j / sigma_hat_j)
assert p_value(30.58, 1.27) < 0.001           # término constante
assert p_value(0.972, 0.103) < 0.001          # num_friends
assert p_value(-1.865, 0.155) < 0.001         # work_hours
assert p_value(0.923, 1.249) > 0.4            # doctorado
```

En una situación diferente a esta, probablemente utilizaríamos software estadístico, que sabe cómo calcular la distribución *t*, además de cómo calcular los errores estándares exactos.

Aunque la mayoría de los coeficientes tienen valores *p* muy pequeños (lo que sugiere que de hecho no son cero), el coeficiente de "doctorado" no es "apreciablemente" distinto de 0, lo que hace probable que el coeficiente de

“doctorado” sea aleatorio en lugar de significativo.

En situaciones de regresión más complicadas, se suele querer probar hipótesis más complejas sobre los datos, como “al menos una de las β_j no es cero” o “β_1 es igual a β_2 y β_3 es igual a β_4”. Esto puede hacerse con una prueba F, pero, por desgracia, eso queda fuera del alcance de este libro.

Regularización

En la práctica, a menudo se suele aplicar regresión lineal a conjuntos de datos con grandes cantidades de variables. Pero esto crea otro par de problemas. Primero, cuantas más variables se empleen, más probable es que se sobreajuste el modelo al conjunto de entrenamiento. Y segundo, cuantos más coeficientes no cero se tengan, más difícil es darles sentido. Si el objetivo es explicar algún fenómeno, un modelo disperso con tres factores podría ser más útil que un modelo ligeramente mejor con cientos.

La regularización es un método en el que añadimos al término de error una penalización, que es mayor a medida que `beta` aumenta. Entonces minimizamos el error y la penalización combinados. Cuanta más importancia le demos al término de penalización, más desalentaremos a los coeficientes grandes.

Por ejemplo, en la regresión *ridge*, añadimos una penalización proporcional a la suma de cuadrados de la `beta_i` (excepto que normalmente no penalizamos `beta_0`, el término constante):

```
# alpha es un *hiperparámetro* que controla lo dura que es la penalización.
# A veces se le llama "lambda" pero eso ya significa algo en Python.
def ridge_penalty(beta: Vector, alpha: float) -> float:
   return alpha * dot(beta[1:], beta[1:])
def squared_error_ridge(x: Vector,
                        y: float,
                        beta: Vector,
                        alpha: float) -> float:
   """estimate error plus ridge penalty on beta"""
   return error(x, y, beta) ** 2 + ridge_penalty(beta, alpha)
```

Podemos después conectar esto con el descenso de gradiente de la manera habitual:

```
from scratch.linear_algebra import add
def ridge_penalty_gradient(beta: Vector, alpha: float) -> Vector:
   """gradient of just the ridge penalty"""
   return [0.] + [2 * alpha * beta_j for beta_j in beta[1:]]
def sqerror_ridge_gradient(x: Vector,
y: float,
beta: Vector,
alpha: float) -> Vector:
   """
   the gradient corresponding to the ith squared error term
   including the ridge penalty
   """
   return add(sqerror_gradient(x, y, beta),
           ridge_penalty_gradient(beta, alpha))
```

Y después tendremos que modificar la función `least_squares_fit` para usar `sqerror_ridge_gradient` en lugar de `sqerror_gradient` (no voy a repetir aquí el código).

Con `alpha` establecido en 0, no hay penalización ninguna y obtenemos los mismos resultados que antes:

```
random.seed(0)
beta_0 = least_squares_fit_ridge(inputs,                    # alpha
daily_minutes_good, 0.0,
               learning_rate, 5000, 25)
# [30.51, 0.97, -1.85, 0.91]
assert 5 < dot(beta_0[1:], beta_0[1:]) < 6
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0) < 0.69
```

A medida que aumentamos `alpha`, la bondad de ajuste empeora, pero el tamaño de `beta` se reduce:

```
beta_0_1 = least_squares_fit_ridge(inputs,                  # alpha
daily_minutes_good, 0.1,
               learning_rate, 5000, 25)
# [30.8, 0.95, -1.83, 0.54]
assert 4 < dot(beta_0_1[1:], beta_0_1[1:]) < 5
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0_1) < 0.69
```

```
beta_1 = least_squares_fit_ridge(inputs,                    # alpha
daily_minutes_good, 1,
               learning_rate, 5000, 25)
# [30.6, 0.90, -1.68, 0.10]
assert 3 < dot(beta_1[1:], beta_1[1:]) < 4
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_1) < 0.69
beta_10 = least_squares_fit_ridge(inputs,                   # alpha
daily_minutes_good,10,
               learning_rate, 5000, 25)
# [28.3, 0.67, -0.90, -0.01]
assert 1 < dot(beta_10[1:], beta_10[1:]) < 2
assert 0.5 < multiple_r_squared(inputs, daily_minutes_good, beta_10) < 0.6
```

En particular, el coeficiente de “doctorado” desaparece a medida que aumentamos la penalización, lo que está de acuerdo con nuestro anterior resultado, que no era apreciablemente distinto de 0.

Nota: Normalmente conviene redimensionar los datos con `rescale` antes de utilizar este método. Después de todo, si cambiáramos años de experiencia por siglos de experiencia, su coeficiente de mínimos cuadrados aumentaría en un factor de 100 y, de repente, resultaría mucho más penalizado, incluso aunque fuera el mismo modelo.

Otro método es la regresión *lasso*, que utiliza la penalización:

```
def lasso_penalty(beta, alpha):
return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

Mientras que la penalización *ridge* encogía en general los coeficientes, la penalización *lasso* tiende a obligar a los coeficientes a ser 0, por lo que nos sirve para aprender modelos dispersos. Lamentablemente, no admite el descenso de gradiente, con lo cual no podremos resolverlo desde cero.

Para saber más

- La regresión tiene una teoría abundante y extensa, por lo que es otro tema sobre el que podría considerar leer un libro de texto, o al menos unos cuantos artículos de la Wikipedia.

- scikit-learn tiene un módulo `linear_model`, en https://scikit-learn.org/stable/modules/linear_model.html, que ofrece un modelo `LinearRegression` similar al nuestro, además de regresión *ridge*, *lasso* y otros tipos de regularización.
- StatsModels, en https://statsmodels.org, es otro módulo de Python que contiene (entre otras cosas) modelos de regresión lineal.

16 Regresión logística

Mucha gente dice que hay una línea muy fina entre el genio y la locura. No creo que haya una línea fina, lo que realmente creo que hay es un abismo.

—Bill Bailey

En el capítulo 1 hemos visto brevemente el problema de tratar de predecir qué usuarios de DataSciencester pagaron por cuentas *premium*. En este capítulo le echaremos otro vistazo a este problema.

El problema

Tenemos un conjunto de datos anónimo de unos 200 usuarios, que contiene el salario de cada usuario, sus años de experiencia como científico de datos y si pagó por una cuenta *premium* (figura 16.1). Como es habitual con las variables categóricas, representamos la variable dependiente como 0 (sin cuenta *premium*) o 1 (con cuenta *premium*).

Como siempre, nuestros datos son una lista de filas `[experience, salary, paid_account]`. Convirtámosla al formato que necesitamos:

```
xs = [[1.0] + row[:2] for row in data]      # [1, experience, salary]
ys = [row[2] for row in data]               # paid_account
```

Un primer intento obvio es utilizar regresión lineal y hallar el mejor modelo:

$$\text{cuenta de pago} = \beta_0 + \beta_1 \text{experiencia} + \beta_2 \text{salario} + \varepsilon$$

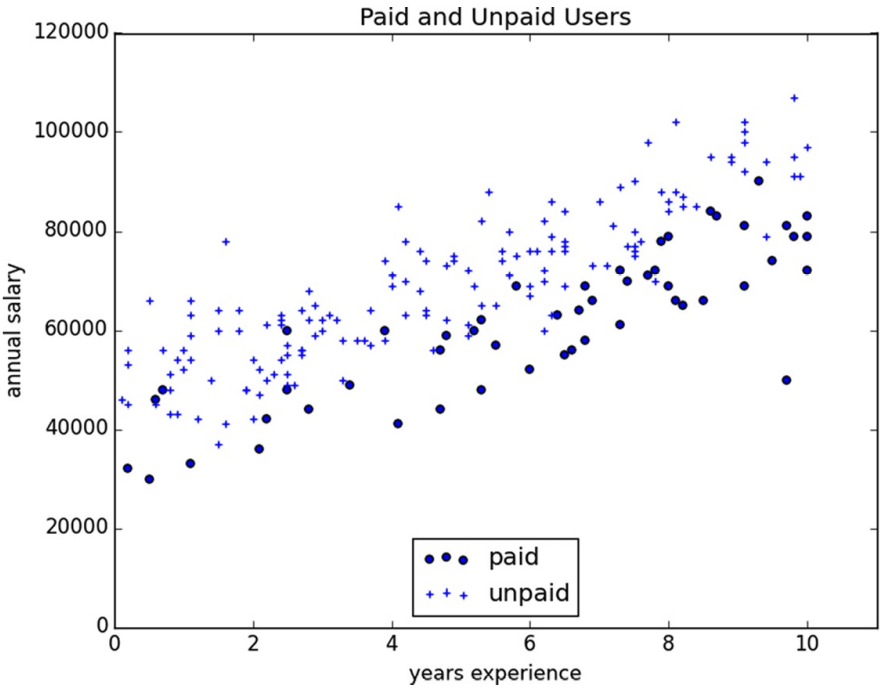


Figura 16.1. Usuarios de pago y no de pago.

Sin duda, no hay nada que nos impida crear así un modelo similar del problema. Los resultados se muestran en la figura 16.2:

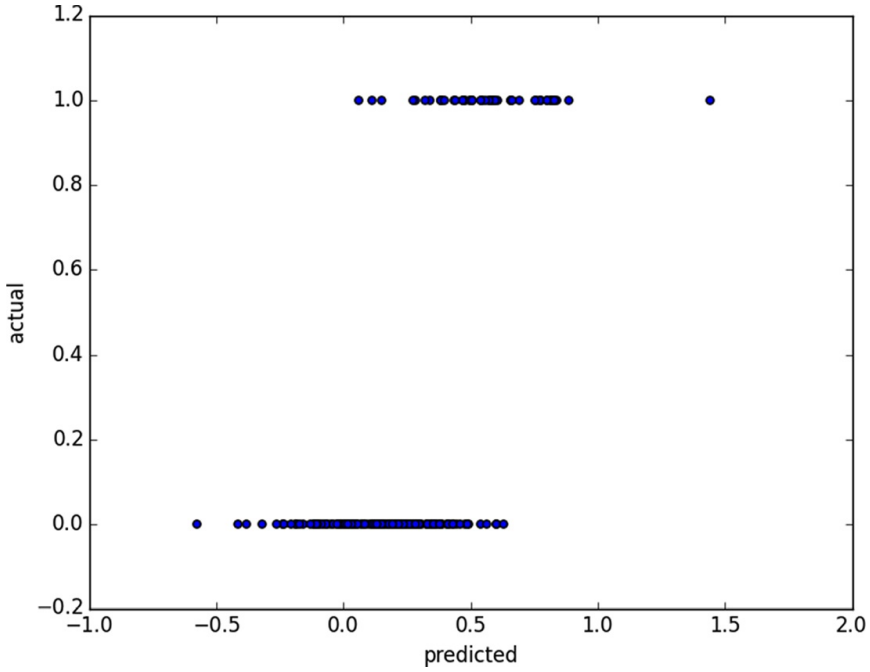


Figura 16.2. Utilizar la regresión lineal para predecir las cuentas de pago.

```
from matplotlib import pyplot as plt
from scratch.working_with_data import rescale
from scratch.multiple_regression import least_squares_fit, predict
from scratch.gradient_descent import gradient_step
learning_rate = 0.001
rescaled_xs = rescale(xs)
beta = least_squares_fit(rescaled_xs, ys, learning_rate, 1000, 1)
# [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_xs]
plt.scatter(predictions, ys)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```

Pero este enfoque nos conduce a un par de problemas inmediatos:

- Nos gustaría que nuestras salidas previstas fueran 0 o 1, para indicar la membresía de clase. Está bien si están entre 0 y 1, ya que podemos interpretarlas como probabilidades (un resultado de 0,25 podría significar un 25 % de posibilidades de ser un miembro de pago). Pero los resultados del modelo lineal pueden ser grandes números positivos o incluso negativos, cuya interpretación no queda clara. De hecho, aquí muchas de nuestras predicciones fueron negativas.
- El modelo de regresión lineal asumía que los errores no estaban correlacionados con las columnas de *x*. Pero, en este caso, el coeficiente de regresión para `experience` es 0,43, indicando que más experiencia da lugar a una mayor probabilidad de una cuenta de pago. Esto significa que nuestro modelo da como resultado valores muy grandes para gente con mucha experiencia. Pero sabemos que los valores reales deben ser como máximo 1, lo que significa que resultados necesariamente muy grandes (y por lo tanto valores muy altos de `experience`) corresponden a valores negativos muy altos del término de error. Como es este el caso, nuestra estimación de `beta` está sesgada.

Lo que realmente nos gustaría es que valores grandes positivos de `dot(x_i, beta)` correspondieran a probabilidades cercanas a 1, y que valores grandes negativos correspondieran a probabilidades cercanas a 0. Podemos

conseguir esto aplicando otra función al resultado.

La función logística

En el caso de la regresión logística, utilizamos la función del mismo nombre, representada en la figura 16.3:

```
def logistic(x: float) -> float:
   return 1.0 / (1 + math.exp(-x))
```

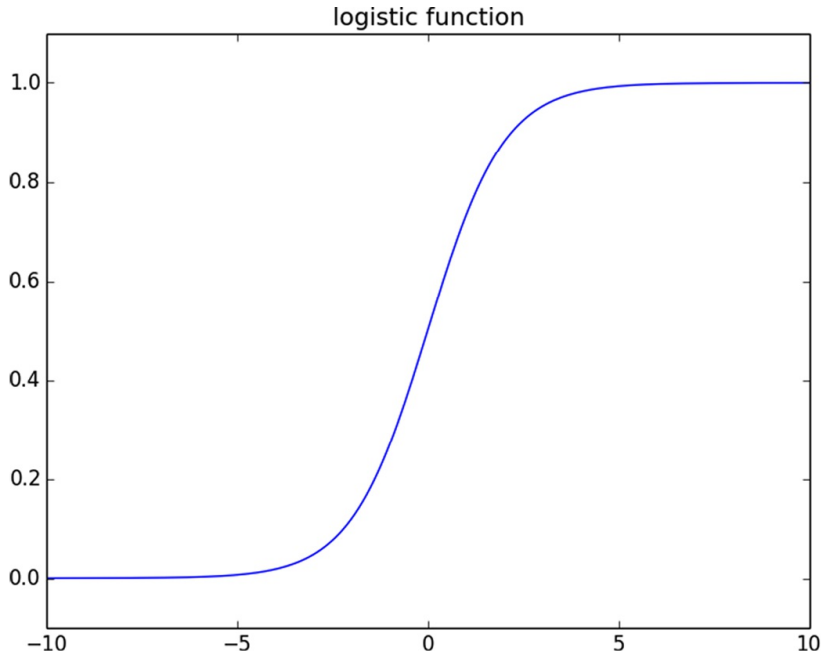


Figura 16.3. La función logística.

A medida que su entrada se hace más grande y positiva, se acerca cada vez más a 1; a medida que se hace más grande y negativa, se va acercando más a 0. Además, tiene la oportuna propiedad de que su derivada viene dada por:

```
def logistic_prime(x: float) -> float:
   y = logistic(x)
   return y * (1 – y)
```

Que utilizaremos en un momento, para ajustar un modelo:

$y_i = f(xi\beta) + \varepsilon_i$

Donde f es la función `logistic`.

Recordemos que para la regresión lineal ajustábamos el modelo minimizando la suma de errores cuadrados, lo que terminaba eligiendo la β que maximizaba la verosimilitud de los datos.

Aquí los dos no son equivalentes, de modo que utilizaremos descenso de gradiente para maximizar la verosimilitud directamente; esto significa que necesitamos calcular la función de verosimilitud y su gradiente.

Dada una cierta β, nuestro modelo dice que cada *yi* debería ser igual a 1 con probabilidad $f(x_i\beta)$ y a 0 con probabilidad 1 - $f(x_i\beta)$.

En particular, la función PDF (de densidad de probabilidad) para *yi* se puede escribir como:

$$p(y_i | x_i, \beta) = f(x_i\beta)^{y_i}(1 - f(x_i\beta))^{1 - y_i}$$

Ya que, si y_i es 0, es igual a:

1 - $f(1)$

Y, si y_i es 1, es igual a:

$f(x_i\beta)$

Resulta que es realmente más sencillo maximizar la log-verosimilitud:

$\log L(\beta \mid x_i, y_i) = y_i \log f(x_i\beta) + (1 - yi) \log (1 - f(x_i\beta))$

Como el logaritmo es una función estrictamente incremental, cualquier `beta` que maximice la log-verosimilitud también hace lo mismo con la verosimilitud, y viceversa. Como el descenso de gradiente minimiza las cosas, realmente trabajaremos con la log-verosimilitud negativa, ya que

maximizar la verosimilitud es lo mismo que minimizar su negativa:

```
import math
from scratch.linear_algebra import Vector, dot

def _negative_log_likelihood(x: Vector, y: float, beta: Vector) -> float:
  """The negative log likelihood for one data point"""
  if y == 1:
    return -math.log(logistic(dot(x, beta)))
  else:
    return -math.log(1–logistic(dot(x, beta)))
```

Si suponemos que los distintos puntos de datos son independientes uno de otro, la verosimilitud total no es más que el producto de las verosimilitudes individuales; lo que significa que la log-verosimilitud total es la suma de las log-verosimilitudes individuales:

```
from typing import List
def negative_log_likelihood(xs: List[Vector],
ys: List[float],
beta: Vector) -> float:
  return sum(_negative_log_likelihood(x, y, beta)
         for x, y in zip(xs, ys))
```

Un poco de cálculo nos da el gradiente:

```
from scratch.linear_algebra import vector_sum
def _negative_log_partial_j(x: Vector, y: float, beta: Vector, j: int) ->
float:
  """
  The jth partial derivative for one data point.
  Here i is the index of the data point.
  """
  return -(y–logistic(dot(x, beta))) * x[j]
def _negative_log_gradient(x: Vector, y: float, beta: Vector) -> Vector:
  """
  The gradient for one data point.
```

```
    """
    return [_negative_log_partial_j(x, y, beta, j)
            for j in range(len(beta))]
def negative_log_gradient(xs: List[Vector],
                          ys: List[float],
                          beta: Vector) -> Vector:
    return vector_sum([_negative_log_gradient(x, y, beta)
                       for x, y in zip(xs, ys)])
```

Momento en el cual tenemos todas las piezas que necesitamos.

Aplicar el modelo

Nos interesará dividir nuestros datos en un conjunto de entrenamiento y uno de prueba:

```
from scratch.machine_learning import train_test_split
import random
import tqdm
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_xs, ys, 0.33)
learning_rate = 0.01
# elige un punto de partida aleatorio
beta = [random.random() for _ in range(3)]
with tqdm.trange(5000) as t:
    for epoch in t:
        gradient = negative_log_gradient(x_train, y_train, beta)
        beta = gradient_step(beta, gradient, -learning_rate)
        loss = negative_log_likelihood(x_train, y_train, beta)
        t.set_description(f"loss: {loss:.3f} beta: {beta}")
```

Y después descubrimos que `beta` es aproximadamente:

```
[-2.0, 4.7, -4.5]
```

Estos son coeficientes para los datos redimensionados con `rescale`, pero también podemos transformarlos de nuevo en los datos originales:

```
from scratch.working_with_data import scale
means, stdevs = scale(xs)
```

```
beta_unscaled = [(beta[0]
             -beta[1] * means[1] / stdevs[1]
             -beta[2] * means[2] / stdevs[2]),
             beta[1] / stdevs[1],
             beta[2] / stdevs[2]]
# [8.9, 1.6, -0.000288]
```

Lamentablemente, estos no son tan fáciles de interpretar como los coeficientes de regresión lineal. Siendo todo lo demás igual, un año de experiencia adicional suma 1,6 a la entrada de `logistic`. Siendo todo lo demás igual, 10.000 dólares extra de salario restan 2,88 a la entrada de `logistic`.

Pero el impacto del resultado depende también de las otras entradas. Si `dot(beta, x_i)` ya es grande (correspondiendo a una probabilidad cercana a 1), aumentarlo aun en una elevada cantidad no puede afectar mucho a la probabilidad. Si está cerca de 0, incrementarlo solo un poquito podría aumentar bastante la probabilidad.

Lo que podemos decir es que (siendo todo lo demás igual) es más probable que la gente con más experiencia pague por las cuentas. Y que (siendo todo lo demás igual) es menos probable que la gente con salarios más altos pague por las cuentas (esto también se hizo evidente al trazar los datos).

Bondad de ajuste

No hemos utilizado aún los datos de prueba que nos quedaban. Veamos lo que ocurre si predecimos cuenta de pago siempre que la probabilidad supere 0,5:

```
true_positives = false_positives = true_negatives = false_negatives = 0
for x_i, y_i in zip(x_test, y_test):
   prediction = logistic(dot(beta, x_i))
   if y_i == 1 and prediction >=          # TP: de pago y predecimos de pago
   0.5:
      true_positives += 1
   elif y_i == 1:                         # FN: de pago y predecimos no de pago
      false_negatives += 1
   elif prediction >= 0.5:                # FP: no de pago y predecimos de pago
```

```
    false_positives += 1
  else:                                # TN: no de pago y predecimos no de
                                       pago
    true_negatives += 1
precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

Esto ofrece una precisión del 75 % (“cuando predecimos cuenta de pago acertamos el 75 % de las veces”) y un recuerdo del 80 % (“cuando un usuario tiene una cuenta de pago predecimos cuenta de pago el 80 % de las veces”), lo que no está nada mal, teniendo en cuenta los pocos datos de que disponemos.

También podemos representar las predicciones frente a los datos reales (figura 16.4), lo que también demuestra que el modelo funciona bien:

```
predictions = [logistic(dot(beta, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test, marker='+')
plt.xlabel("predicted probability")
plt.ylabel("actual outcome")
plt.title("Logistic Regression Predicted vs. Actual")
plt.show()
```

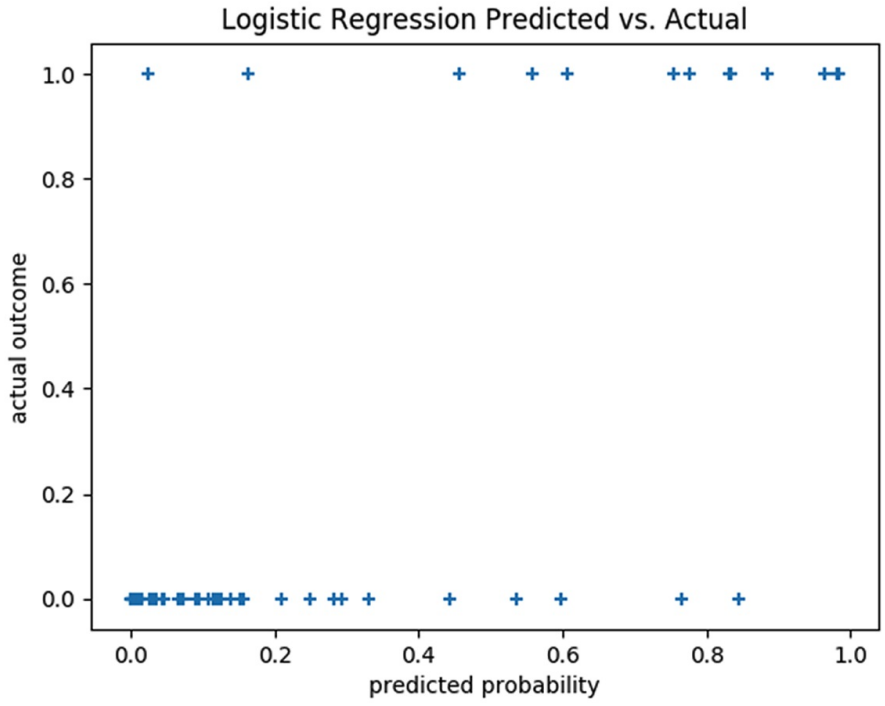


Figura 16.4. Regresión logística predicha frente a real.

Máquinas de vectores de soporte

El conjunto de puntos donde `dot(beta, x_i)` es igual a 0 es la frontera entre nuestras clases. Podemos representar esto para ver exactamente lo que está haciendo nuestro modelo (figura 16.5).

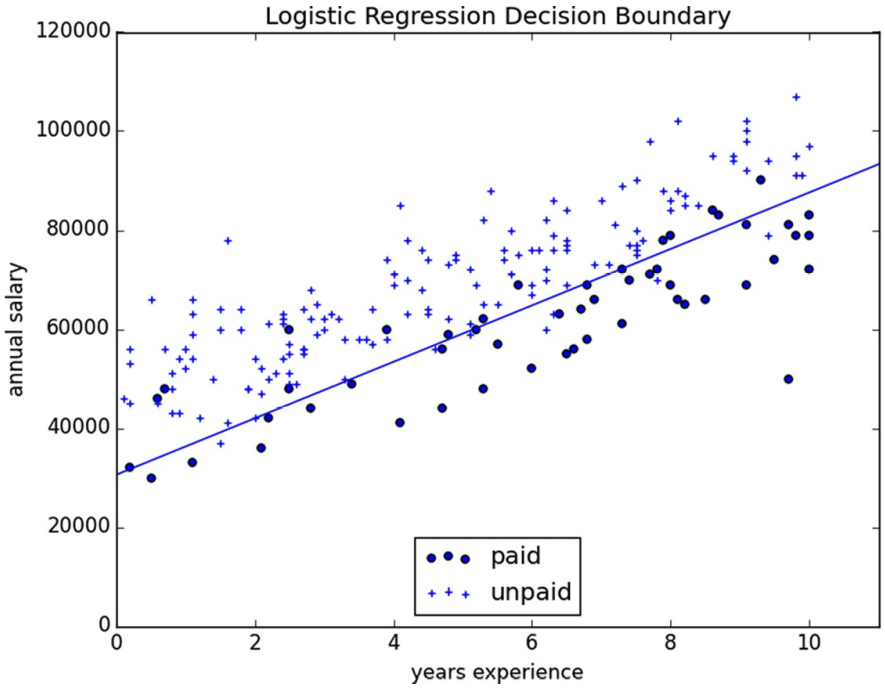


Figura 16.5. Usuarios con cuentas de pago y no de pago con frontera de decisión.

Esta frontera es un hiperplano, que divide el espacio de parámetros en dos mitades correspondientes a predice de pago y predice no de pago. Lo descubrimos como un efecto colateral de hallar el modelo logístico más probable.

Un método alternativo a la clasificación es simplemente buscar el hiperplano que "mejor" separe las clases en los datos de entrenamiento. Esta es la idea de la máquina de vectores de soporte, que localiza el hiperplano que maximiza la distancia al punto más cercano en cada clase (figura 16.6).

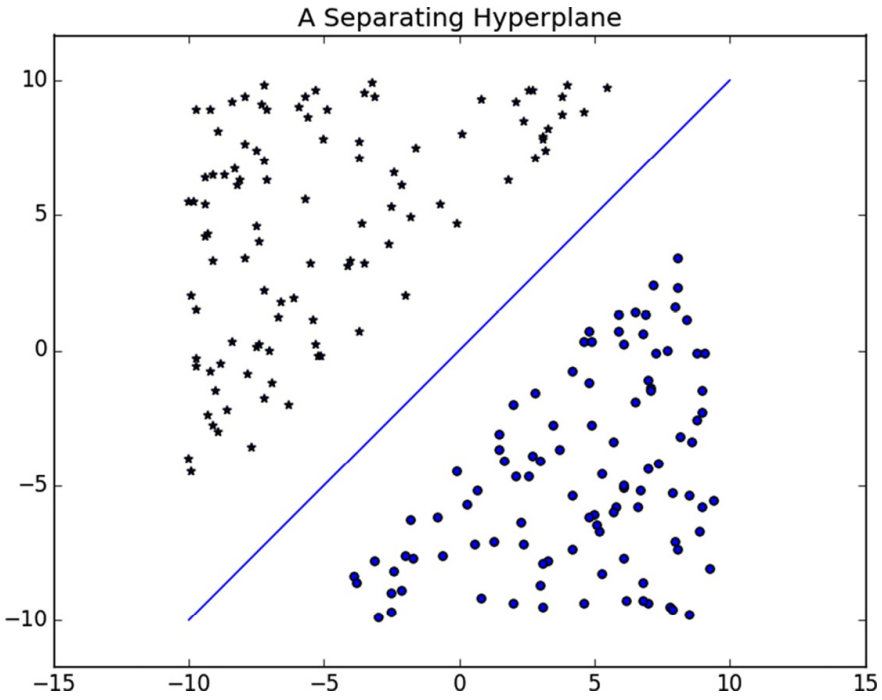


Figura 16.6. Un hiperplano de separación.

Encontrar un hiperplano como este es un problema de optimización que implica técnicas demasiado avanzadas para nosotros. Un problema distinto es que un hiperplano de separación podría no existir. En nuestro conjunto de datos "¿quién paga?", simplemente es que no hay línea que separe perfectamente los usuarios que pagan de los que no pagan.

En ocasiones, podemos sortear esto transformando los datos en un espacio de muchas dimensiones. Por ejemplo, veamos el sencillo conjunto de datos unidimensional mostrado en la figura 16.7.

Sin duda, no hay hiperplano que separe los ejemplos positivos de los negativos. Sin embargo, veamos lo que ocurre cuando mapeamos este conjunto de datos en dos dimensiones enviando el punto `x` a `(x, x**2)`. De repente es posible encontrar un hiperplano que divida los datos (figura 16.8).

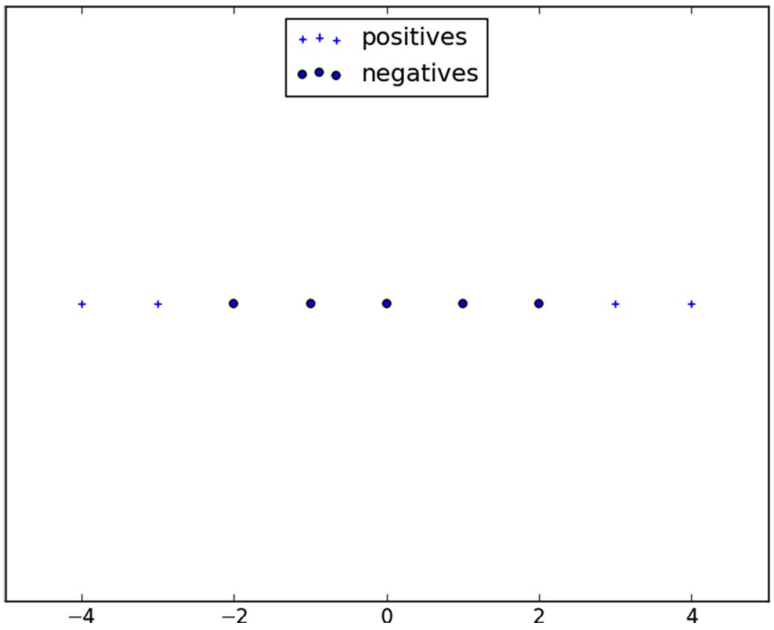


Figura 16.7. Un conjunto de datos unidimensional no separable.

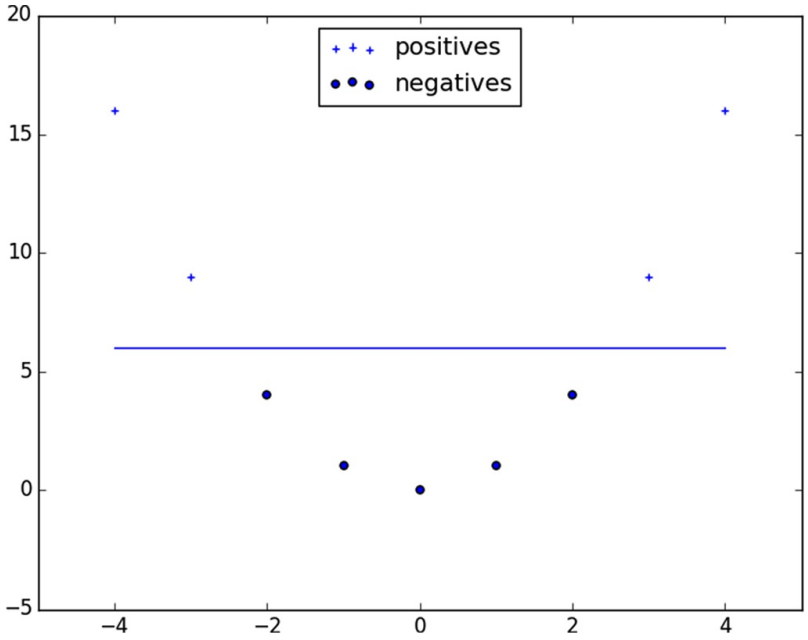


Figura 16.8. El conjunto de datos se vuelve separable con muchas dimensiones.

Esto se denomina el truco del *kernel*, porque, en lugar de mapear realmente los puntos en el espacio de muchas dimensiones (lo que podría resultar caro si hay muchos puntos y el mapeado es complicado), podemos utilizar una función "*kernel*" para calcular productos de punto en el espacio

de muchas dimensiones y utilizarlos para encontrar un hiperplano.

Es difícil (y probablemente en absoluto una buena idea) utilizar máquinas de vectores de soporte sin confiar en software de optimización especializado escrito por gente con la experiencia adecuada, de modo que tendremos que dejar aquí nuestro planteamiento.

Para saber más

- scikit-learn tiene módulos de regresión logística, en https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression, y de máquinas de vectores de soporte, en https://scikit-learn.org/stable/modules/svm.html.
- LIBSVM, en https://www.csie.ntu.edu.tw/~cjlin/libsvm/, es la implementación de máquina de soporte vectorial que realmente utiliza scikit-learn. Su sitio web tiene mucha documentación variada sobre este algoritmo.

17 Árboles de decisión

Un árbol es un misterio incomprensible.

—Jim Woodring

El vicepresidente de Talento de DataSciencester ha entrevistado a varios candidatos del sitio para un puesto de trabajo, con diversos grados de éxito. Ha recogido un conjunto de datos, que consiste en varios atributos (cualitativos) de cada candidato, además de si la entrevista con cada uno fue bien o mal. Así que plantea la siguiente pregunta: ¿se podrían utilizar estos datos para crear un modelo que identifique los candidatos que harán una buena entrevista, de forma que no tenga que perder el tiempo en esta tarea?

Parece que en esto encaja bien un árbol de decisión, otra herramienta de creación de modelos predictivos que forma parte del equipo del científico de datos.

¿Qué es un árbol de decisión?

Un árbol de decisión emplea una estructura en árbol para representar una serie de posibles rutas de ramificación y un resultado para cada ruta.

Si alguna vez ha jugado al juego de las 20 preguntas,[1] estará familiarizado con los árboles de decisión. Por ejemplo:

- "Estoy pensando en un animal".
- "¿Tiene más de cinco patas?".
- "No".
- "¿Es delicioso?".
- "No".
- "¿Aparece en el reverso de la moneda de cinco centavos australiana?".
- "Sí".

- “¿Es un equidna?”.
- “¡Sí, correcto!”.

Esta batería de preguntas corresponde a la siguiente ruta, que sería la de un árbol de decisión “adivine el animal” bastante singular (y no muy amplio) (figura 17.1):

“No más de 5 patas” → “No delicioso” → “En la moneda de 5 centavos” → “¡Equidna!”

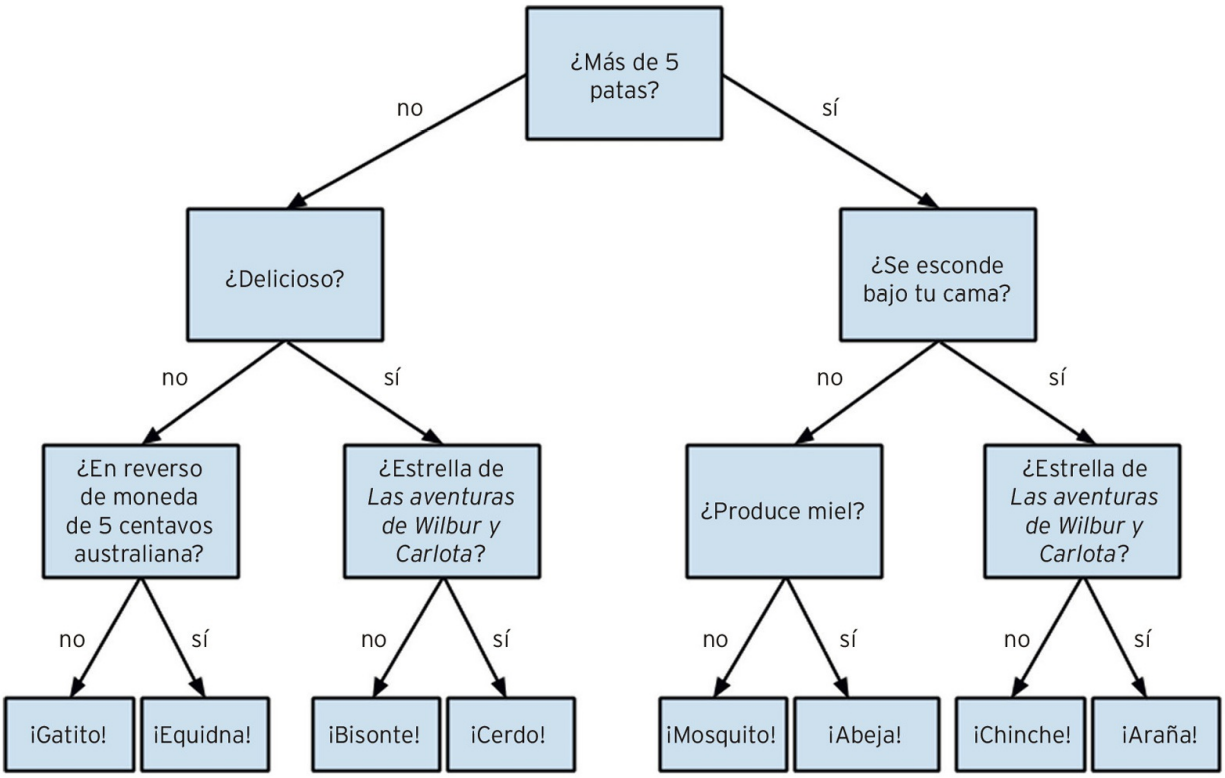


Figura 17.1. Un árbol de decisión “adivine el animal”.

Los árboles de decisión son recomendables por varias razones. Son muy fáciles de entender e interpretar, y el proceso mediante el cual alcanzan una predicción es totalmente transparente. A diferencia de los otros modelos que hemos visto hasta ahora, los árboles de decisión pueden gestionar sin problemas una mezcla de atributos numéricos (por ejemplo, número de patas) y categóricos (por ejemplo, delicioso/no delicioso) y pueden incluso

clasificar datos para los que falten atributos.

Al mismo tiempo, encontrar un árbol de decisión "óptimo" para un conjunto de datos de entrenamiento es un problema muy complicado desde el punto de vista computacional (solucionaremos esto tratando de crear un árbol bastante bueno en lugar de óptimo, aunque para conjuntos de datos grandes puede suponer mucho trabajo). Aún más importante es el hecho de que es muy fácil (y muy malo) crear árboles de decisión que estén sobreajustados a los datos de entrenamiento y que no generalicen bien con datos no visibles. Veremos formas de resolver esto.

La mayoría de la gente divide los árboles de decisión en árboles de clasificación (que producen resultados categóricos) y árboles de regresión (que producen resultados numéricos). En este capítulo, nos centraremos en los árboles de clasificación y estudiaremos el algoritmo ID3 para lograr un árbol de decisión a partir de un conjunto de datos etiquetados, lo que debería permitirnos entender cómo funcionan realmente estos algoritmos. Para simplificar las cosas, nos limitamos a problemas con resultados binarios como "¿debería contratar a este candidato?", "¿debería mostrar al visitante de este sitio web el anuncio A o el anuncio B?" o "¿me enfermaré si me como esta comida que encontré en la nevera de la oficina?".

Entropía

Para crear un árbol de decisión, necesitaremos decidir qué preguntas formular y en qué orden. En cada etapa del árbol hay algunas posibilidades que hemos eliminado y otras que no. Tras descubrir que un animal no tiene más de cinco patas, hemos eliminado la posibilidad de que sea un saltamontes. No hemos eliminado la posibilidad de que sea un pato. Cada posible pregunta divide las posibilidades restantes según su respuesta.

Lo ideal sería elegir preguntas cuyas respuestas dieran mucha información sobre lo que debería predecir nuestro árbol. Si hay una sola pregunta sí/no para la que las respuestas "sí" siempre corresponden a resultados `True` y las respuestas "no" a resultados `False` (o viceversa), sería la pregunta perfecta.

Pero, sin embargo, probablemente no sería una buena opción una pregunta sí/no para la que ninguna respuesta dé mucha información nueva sobre cómo debería ser la predicción.

Esta noción de "cantidad de información" se captura con la entropía. Es probable que haya oído ya antes este término con el significado de desorden. Aquí lo utilizamos para representar la incertidumbre asociada a los datos.

Supongamos que tenemos un conjunto S de datos, cada miembro del cual está etiquetado como perteneciente a una de un número finito de clases C_1, ..., C_n. Si todos los puntos de datos pertenecen a una sola clase, entonces no hay incertidumbre, con lo cual idealmente la entropía sería baja. Si los puntos de datos están distribuidos por igual a lo largo de las clases, sí habría mucha incertidumbre y la entropía sería alta.

En términos matemáticos, si p_i es la proporción de datos etiquetados como clase c_i, definimos la entropía como:

$$H(S) = -p_1 \log_2 p_1 - ... - p_n \log_2 p_n$$

Con el convenio (estándar) de que $0 \log 0 = 0$.

Sin preocuparnos demasiado por los detalles, cada término $-p_i \log_2 p_i$ es no negativo y está cerca de 0 precisamente cuando p_i está o bien cerca de 0 o cerca de 1 (figura 17.2).

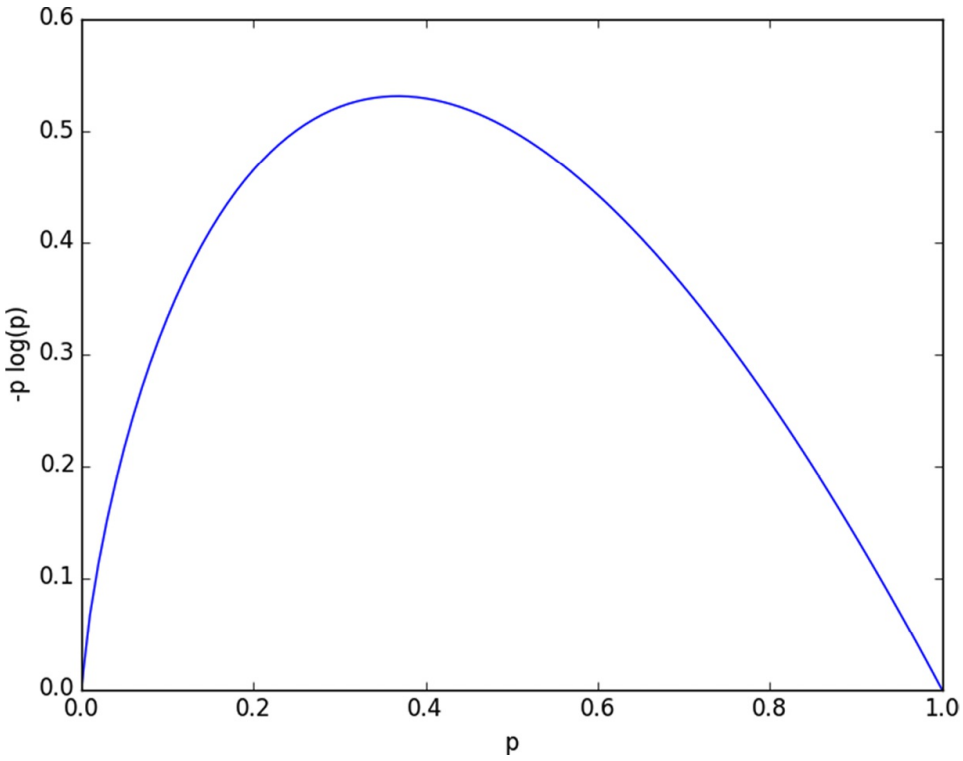


Figura 17.2. Una representación de – *p* log *p*.

Esto significa que la entropía será pequeña cuando cada p_i esté cerca de 0 o 1 (es decir, cuando la mayoría de los datos están en una sola clase), y será más grande cuando muchos de los p_i no estén cerca de 0 (es decir, cuando los datos estén repartidos a lo largo de varias clases). Este es exactamente el comportamiento que deseamos.

Es bastante sencillo desarrollar todo esto en una función:

```
from typing import List
import math
def entropy(class_probabilities: List[float]) -> float:
   """Given a list of class probabilities, compute the entropy"""
   return sum(-p * math.log(p, 2)
           for p in class_probabilities
           if p > 0)          # ignora probabilidades cero
assert entropy([1.0]) == 0
assert entropy([0.5, 0.5]) == 1
assert 0.81 < entropy([0.25, 0.75]) < 0.82
```

Nuestros datos consistirán en pares `(input, label)`, lo que significa que

tendremos que calcular nosotros mismos las probabilidades de clase. Hay que tener en cuenta que no nos preocupa realmente qué etiqueta está asociada a qué probabilidad, únicamente cuáles son las probabilidades:

```
from typing import Any
from collections import Counter
def class_probabilities(labels: List[Any]) -> List[float]:
   total_count = len(labels)
   return [count / total_count
        for count in Counter(labels).values()]
def data_entropy(labels: List[Any]) -> float:
   return entropy(class_probabilities(labels))
assert data_entropy([‘a’]) == 0
assert data_entropy([True, False]) == 1
assert data_entropy([3, 4, 4, 4]) == entropy([0.25, 0.75])
```

La entropía de una partición

Lo que hemos hecho hasta ahora es calcular la entropía (o sea, “incertidumbre”) de un conjunto único de datos etiquetados. Pero cada etapa de un árbol de decisión implica formular una pregunta cuya respuesta reparte los datos en uno o (supuestamente) varios subconjuntos. Por ejemplo, nuestra pregunta “¿tiene más de cinco patas?” divide los animales en los que tienen más de cinco patas (por ejemplo, las arañas) y los que no (por ejemplo, los equidnas).

En consecuencia, queremos tener una cierta noción de la entropía resultante de la partición de un conjunto de datos de una determinada forma. Queremos que una partición tenga entropía baja si reparte los datos en subconjuntos que tienen también entropía baja (es decir, son muy seguros), y entropía alta si contiene subconjuntos que (son grandes y) tienen asimismo entropía alta (es decir, son muy inciertos).

Por ejemplo, la pregunta de la “moneda de cinco céntimos australiana” era bastante tonta (¡aunque también bastante afortunada!), ya que dividió los animales que quedaban en ese punto en S_1 = {equidna} y S_2 = {los demás}, donde S_2 es grande y además tiene alta entropía (S_1 no tiene entropía, pero

representa una pequeña fracción de las “clases” restantes).

Matemáticamente, si dividimos nuestros datos S en subconjuntos S_1, ..., S_m conteniendo proporciones q_1, ..., q_m de los datos, calculamos entonces la entropía de la partición como una suma ponderada:

$H = q_1H(S_1) + ... + q_mH(S_m)$

Que podemos implementar como:

```
def partition_entropy(subsets: List[List[Any]]) -> float:
   """Returns the entropy from this partition of data into subsets"""
   total_count = sum(len(subset) for subset in subsets)
   return sum(data_entropy(subset) * len(subset) / total_count
           for subset in subsets)
```

Nota: Un problema con este planteamiento es que repartir según un atributo con muchos valores distintos dará como resultado una entropía muy baja debido al sobreajuste. Por ejemplo, imaginemos que trabajamos en un banco y estamos tratando de crear un árbol de decisión para predecir cuál de sus clientes es probable que no pague la hipoteca, utilizando algunos datos históricos, como el conjunto de entrenamiento de que disponemos. Vayamos aún más allá y supongamos que el conjunto de datos contiene el número de la Seguridad Social de cada cliente. Dividir según el NSS producirá subconjuntos de una sola persona, cada uno de los cuales tiene necesariamente cero entropía. Pero podemos tener la certeza de que un modelo que se base en el NSS no generaliza más allá del conjunto de entrenamiento. Por esta razón, al crear árboles de decisión deberíamos probablemente tratar de evitar (o poner en *buckets*, si corresponde) atributos con grandes cantidades de valores posibles.

Crear un árbol de decisión

El vicepresidente le ofrece los datos del entrevistado, que consisten en (según su especificación) un módulo `NamedTuple` de los atributos más importantes de cada candidato: su nivel (*level*), su lenguaje predilecto (*lang*), si es activo o no en Twitter (*tweets*), si tiene doctorado (*phd*) y si la entrevista fue positiva (*did_well*):

```
from typing import NamedTuple, Optional
class Candidate(NamedTuple):
   level: str
   lang: str
   tweets: bool
   phd: bool
   did_well: Optional[bool] = None      # permite datos sin etiquetar
             # level lang tweets phd did_well
inputs = [Candidate('Senior', 'Java', False, False, False),
        Candidate('Senior', 'Java', False, True, False),
        Candidate('Mid', 'Python', False, False, True),
        Candidate('Junior', 'Python', False, False, True),
        Candidate('Junior', 'R', True, False, True),
        Candidate('Junior', 'R', True, True, False),
        Candidate('Mid', 'R', True, True, True),
        Candidate('Senior', 'Python', False, False, False),
        Candidate('Senior', 'R', True, False, True),
        Candidate('Junior', 'Python', True, False, True),
        Candidate('Senior', 'Python', True, True, True),
        Candidate('Mid', 'Python', False, True, True),
        Candidate('Mid', 'Java', True, False, True),
        Candidate('Junior', 'Python', False, True, False)
        ]
```

Nuestro árbol consiste en nodos de decisión (que formulan una pregunta y nos dirigen de forma diferente dependiendo de la respuesta) y nodos de hoja (que nos dan una predicción). Lo crearemos utilizando el algoritmo ID3, relativamente sencillo, que funciona del siguiente modo. Digamos que nos han dado datos etiquetados y una lista de atributos para considerar las ramificaciones:

- Si los datos tienen todos la misma etiqueta, crea un nodo de hoja que predice la etiqueta y después se detiene.
- Si la lista de atributos está vacía (es decir, no hay más preguntas posibles que formular), crea un nodo de hoja que predice la etiqueta más habitual y después se detiene.
- Si no, intenta dividir los datos por cada uno de los atributos.
- Elige la bifurcación con la entropía más baja posible.
- Añade un nodo de decisión basándose en el atributo elegido.
- Vuelve a repetirlo en cada subconjunto dividido utilizando los atributos restantes.

Esto es lo que se conoce como algoritmo “voraz” porque, en cada paso, elige la mejor opción inmediata. Dado un conjunto de datos, puede haber un árbol mejor con un primer movimiento de peor aspecto. Si es así, este algoritmo no lo encontrará. No obstante, es relativamente fácil de comprender e implementar, por lo que nos ofrece un buen punto de partida para empezar a explorar los árboles de decisión.

Vayamos manualmente por cada uno de estos pasos en el conjunto de datos del entrevistado. El conjunto tiene ambas etiquetas `True` y `False`, y tenemos cuatro atributos según los cuales podemos repartir. Así que nuestro primer paso será hallar la división con la menor entropía. Empezaremos escribiendo una función que se encarga del proceso de partición:

```
from typing import Dict, TypeVar
from collections import defaultdict
T = TypeVar('T')                    # tipo genérico para entradas
def partition_by(inputs: List[T], attribute: str) -> Dict[Any, List[T]]:
   """Partition the inputs into lists based on the specified attribute."""
   partitions: Dict[Any, List[T]] = defaultdict(list)
   for input in inputs:
      key = getattr(input,                # valor del atributo especificado
      attribute)
      partitions[key].append(input)       # añade entrada a la partición
                                          correcta
   return partitions
```

Y otra que la utilice para calcular la entropía:

```
def partition_entropy_by(inputs: List[Any],
                  attribute: str,
                  label_attribute: str) -> float:
   """Compute the entropy corresponding to the given partition"""
   # partitions consiste en nuestras entradas
   partitions = partition_by(inputs, attribute)
   # pero partition_entropy solo necesita las etiquetas de clase
   labels = [[getattr(input, label_attribute) for input in partition]
           for partition in partitions.values()]
   return partition_entropy(labels)
```

Ahora solo necesitamos la partición de mínima entropía para el conjunto

de datos entero:

```
for key in ['level','lang','tweets','phd']:
   print(key, partition_entropy_by(inputs, key, 'did_well'))
assert 0.69 < partition_entropy_by(inputs, 'level', 'did_well') < 0.70
assert 0.86 < partition_entropy_by(inputs, 'lang', 'did_well') < 0.87
assert 0.78 < partition_entropy_by(inputs, 'tweets', 'did_well') < 0.79
assert 0.89 < partition_entropy_by(inputs, 'phd', 'did_well') < 0.90
```

La entropía más baja viene de dividir según `level`, de modo que nos hará falta hacer un subárbol para cada posible valor de `level`. Cada candidato `Mid` se etiqueta `True`, lo que significa que el subárbol `Mid` es simplemente un nodo de hoja que predice `True`. Para candidatos `Senior`, tenemos una mezcla de valores `True` y `False`, de modo que tenemos que dividir de nuevo:

```
senior_inputs = [input for input in inputs if input.level == 'Senior']
assert 0.4 == partition_entropy_by(senior_inputs, 'lang', 'did_well')
assert 0.0 == partition_entropy_by(senior_inputs, 'tweets', 'did_well')
assert 0.95 < partition_entropy_by(senior_inputs, 'phd', 'did_well') < 0.96
```

Esto nos demuestra que nuestra siguiente división debería realizarse según `tweets`, lo que da como resultado una partición con entropía cero. Para esos candidatos de nivel `Senior`, un "sí" en el atributo de los tuits siempre da como resultado `True`, mientras que un "no" siempre da como resultado `False`.

Por último, si hacemos lo mismo para los candidatos `Junior`, terminamos repartiendo según `phd`, tras de lo cual descubrimos que no tener doctorado siempre da como resultado `True` y tener doctorado siempre da `False`.

La figura 17.3 muestra el árbol de decisión completo.

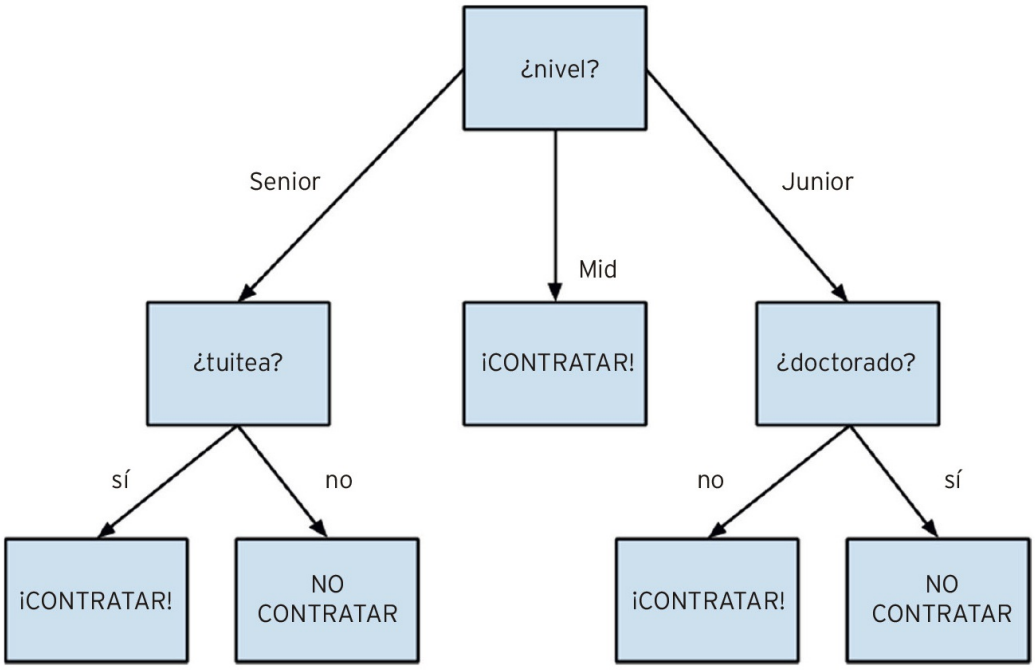


Figura 17.3. El árbol de decisión para contrataciones.

Ahora, a combinarlo todo

Ahora que ya hemos visto cómo funciona el algoritmo, nos gustaría implementarlo más en general, lo que significa que tenemos que decidir cómo queremos representar los árboles. Utilizaremos la representación más liviana posible. Definimos un árbol de dos maneras:

- Un `Leaf` (que predice un solo valor).
- Un `Split` (que contiene un atributo según el que dividir, subárboles para valores determinados de ese atributo y posiblemente un valor predeterminado que utilizar si encontramos un valor desconocido).

```
from typing import NamedTuple, Union, Any
class Leaf(NamedTuple):
   value: Any
class Split(NamedTuple):
   attribute: str
   subtrees: dict
```

```
  default_value: Any = None
DecisionTree = Union[Leaf, Split]
```

Con esta representación, nuestro árbol de contrataciones tendría este aspecto:

```
hiring_tree = Split('level', {      # primero, considera "level"
  'Junior': Split('phd', {          # si el nivel es "Junior", mira "phd"
    False: Leaf(True),              # si "phd" es False, predice True
    True: Leaf(False)               # si "phd" es True, predice False
  }),
  'Mid': Leaf(True),              # si el nivel es "Mid", predice True
  'Senior': Split('tweets', {       # si el nivel es "Senior", mira "tweets"
    False: Leaf(False),             # si "tweets" es False, predice False
    True: Leaf(True)                # si "tweets" es True, predice True
  })
})
```

Queda pendiente la cuestión de qué hacer si encontramos un valor de atributo inesperado (o faltante). ¿Qué debe hacer nuestro árbol de contrataciones si encuentra un candidato cuyo `level` es `Intern`? Gestionaremos este caso asignando al atributo `default_value` la etiqueta más común.

Dada una representación como esta, podemos clasificar una entrada con:

```
def classify(tree: DecisionTree, input: Any) -> Any:
"""classify the input using the given decision tree"""
  # Si es un nodo de hoja, devuelve su valor
  if isinstance(tree, Leaf):
    return tree.value
  # Si no este árbol consiste en un atributo según el que dividir
  # y un diccionario cuyas claves son valores de ese atributo
  # y cuyos valores son subárboles que considerar después
  subtree_key = getattr(input, tree.attribute)
  if subtree_key not in                  # Si no hay subárbol para clave,
  tree.subtrees:
    return tree.default_value            # devuelve el valor predeterminado.
  subtree =                              # Elige el subárbol adecuado
  tree.subtrees[subtree_key]
  return classify(subtree, input)        # y lo usa para clasificar la
                                         entrada.
```

Todo lo que queda por hacer es crear la representación del árbol a partir de nuestros datos de entrenamiento:

```
def build_tree_id3(inputs: List[Any],
             split_attributes: List[str],
             target_attribute: str) -> DecisionTree:
     # Cuenta etiquetas destino
     label_counts = Counter(getattr(input, target_attribute)
for input in inputs)
most_common_label = label_counts.most_common(1)[0][0]
   # Si hay una etiqueta única, la predice
   if len(label_counts) == 1:
      return Leaf(most_common_label)
   # Si no quedan atributos de división, devuelve la etiqueta mayoritaria
   if not split_attributes:
      return Leaf(most_common_label)
   # Si no divide por el mejor atributo
   def split_entropy(attribute: str) -> float:
      """Helper function for finding the best attribute"""
      return partition_entropy_by(inputs, attribute, target_attribute)
   best_attribute = min(split_attributes, key=split_entropy)
   partitions = partition_by(inputs, best_attribute)
   new_attributes = [a for a in split_attributes if a != best_attribute]
   # Crea recursivamente los subárboles
   subtrees = {attribute_value : build_tree_id3(subset,
 new_attributes,
 target_attribute)
           for attribute_value, subset in partitions.items()}
   return Split(best_attribute, subtrees, default_value=most_common_label)
```

En el árbol que creamos, cada hoja estaba enteramente formada por entradas `True` o por entradas `False`. Esto significa que el árbol predice perfectamente según el conjunto de datos de entrenamiento. Pero también podemos aplicarlo a nuevos datos que no estuvieran en el conjunto de entrenamiento:

```
tree = build_tree_id3(inputs,
                ['level', 'lang', 'tweets', 'phd'],
                'did_well')
# Debe predecir True
assert classify(tree, Candidate("Junior", "Java", True, False))
# Debe predecir False
```

```
assert not classify(tree, Candidate("Junior", "Java", True, True))
```

Y también a datos con valores inesperados:

```
# Debe predecir True
assert classify(tree, Candidate("Intern", "Java", True, True))
```

Nota: Como nuestro objetivo era principalmente mostrar cómo crear un árbol, lo construimos utilizando el conjunto de datos entero. Como siempre, si estuviéramos tratando realmente de crear un buen modelo para algo, habríamos recogido más datos y los habríamos dividido en subconjuntos de entrenamiento/validación/prueba.

Bosques aleatorios

Teniendo en cuenta lo mucho que pueden los árboles de decisión ajustarse a sus datos de entrenamiento, no resulta sorprendente que tengan tendencia a sobreajustar. Una forma de evitar esto es una técnica llamada bosques aleatorios, en la que creamos varios árboles de decisión y combinamos sus resultados. Si son árboles de clasificación, podríamos dejarlos votar; si son de regresión, podríamos promediar sus predicciones.

Nuestro proceso de creación de árboles era determinista, de modo que ¿cómo obtenemos árboles aleatorios?

Una parte del proceso implica aplicar *bootstrap* a los datos (recordemos la sección sobre *bootstrap* en el capítulo 15). En lugar de entrenar cada árbol en todas las entradas del conjunto de entrenamiento, lo entrenamos en el resultado de `bootstrap_sample (inputs)`. Como cada árbol se ha creado usando distintos datos, cada uno será diferente de los demás (un beneficio secundario es que es totalmente justo utilizar los datos no muestreados para probar cada árbol, lo que significa que uno se puede salir con la suya utilizando todos los datos como conjunto de entrenamiento si se es lo bastante listo midiendo el rendimiento). Esta técnica se conoce como agregación o empaquetado de *bootstrap*.

Una segunda fuente de aleatoriedad implica cambiar la forma que tenemos

de elegir el `best_attribute` según el cual dividir. En lugar de mirar todos los atributos restantes, primero elegimos un subconjunto aleatorio de ellos y después repartimos según el de ellos que sea mejor:

```
  # si ya hay suficientes candidatos divididos, los mira todos
  if len(split_candidates) <= self.num_split_candidates:
     sampled_split_candidates = split_candidates
  # si no elige una muestra aleatoria
  else:
     sampled_split_candidates = random.sample(split_candidates,
self.num_split_candidates)
  # ahora elige el mejor atributo solo de esos candidatos
  best_attribute = min(sampled_split_candidates, key=split_entropy)
  partitions = partition_by(inputs, best_attribute)
```

Este es un ejemplo de una técnica más amplia llamada aprendizaje combinado o *ensemble*, en la que combinamos varios estudiantes débiles (normalmente modelos de alto sesgo y baja varianza) para producir un modelo global fuerte.

Para saber más

- scikit-learn tiene muchos modelos de árbol de decisión, en https://scikit-learn.org/stable/modules/tree.html. También tiene un módulo `ensemble` en https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble, que incluye un `RandomForestClassifier`, además de otros métodos de aprendizaje combinado.
- XGBoost, en https://xgboost.ai/, es una librería para entrenar árboles de decisión con potenciación del gradiente que tiende a ganar muchas competiciones de *machine learning* de estilo Kaggle.
- Apenas hemos arañado la superficie de los árboles de decisión y sus algoritmos. La Wikipedia, en https://es.wikipedia.org/wiki/Aprendizaje_basado_en_%C3%A1rb es un buen punto de partida para un estudio más detallado.

[1] https://en.wikipedia.org/wiki/Twenty_questions.

18 Redes neuronales

Me gusta el sinsentido; despierta las células del cerebro.

—Dr. Seuss

Una red neuronal artificial (o red neuronal sin más) es un modelo predictivo motivado por el modo en que funciona el cerebro. Piense en el cerebro como en una colección de neuronas conectadas entre ellas. Cada neurona mira las salidas de las otras neuronas que la alimentan, hace un cálculo y después se activa (si el cálculo excede un cierto umbral) o no (si no lo excede).

Según esta explicación, las redes neuronales artificiales están formadas por neuronas artificiales, que realizan cálculos similares sobre sus entradas. Las redes neuronales pueden resolver distintos problemas, como reconocimiento de escritura y detección de caras, y se utilizan mucho en el *deep learning*, uno de los subcampos más recientes de la ciencia de datos. Sin embargo, la mayoría de las redes neuronales son "cajas negras" (es decir, inspeccionar sus detalles no permite entender mucho mejor cómo resuelven un problema). Además, pueden ser difíciles de entrenar. Para resolver la mayor parte de los problemas que uno se suele encontrar como científico de datos en ciernes, probablemente no son la mejor opción. Pero, si algún día el objetivo es construir una inteligencia artificial para crear la singularidad, entonces sí podrían ser de utilidad.

Perceptrones

La red neuronal más sencilla de todas es el perceptrón, que se aproxima a una sola neurona con *n* entradas binarias. Calcula una suma ponderada de sus entradas y se "activa" si esa suma es 0 o mayor que 0:

```
from scratch.linear_algebra import Vector, dot
```

```
def step_function(x: float) -> float:
   return 1.0 if x >= 0 else 0.0
def perceptron_output(weights: Vector, bias: float, x: Vector) -> float:
   """Returns 1 if the perceptron 'fires', 0 if not"""
   calculation = dot(weights, x) + bias
   return step_function(calculation)
```

El perceptrón simplemente distingue entre los semiespacios separados por el hiperplano de puntos `x`, para los cuales:

```
dot(weights, x) + bias == 0
```

Con pesos adecuadamente elegidos, los perceptrones pueden resolver unos cuantos problemas sencillos (véase la figura 18.1). Por ejemplo, podemos crear una puerta AND, que devuelve 1 si sus dos entradas son 1 y 0 si una de sus entradas es 0, utilizando:

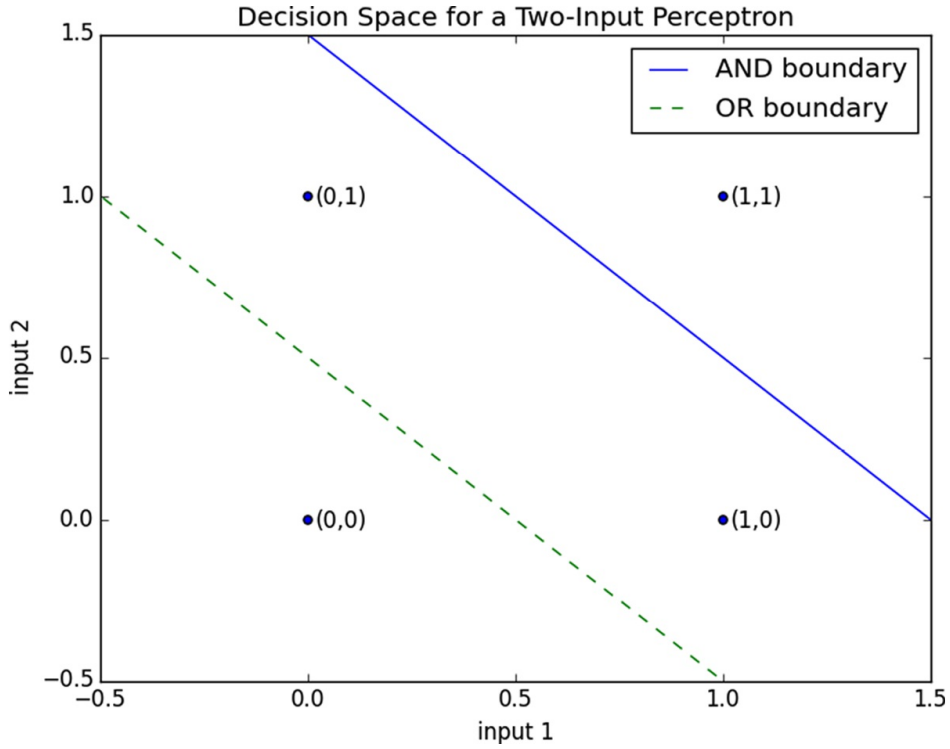


Figura 18.1. Espacio de decisión para un perceptrón de dos entradas.

```
and_weights = [2., 2]
and_bias = -3.
assert perceptron_output(and_weights, and_bias, [1, 1]) == 1
assert perceptron_output(and_weights, and_bias, [0, 1]) == 0
```

```
assert perceptron_output(and_weights, and_bias, [1, 0]) == 0
assert perceptron_output(and_weights, and_bias, [0, 0]) == 0
```

Si ambas entradas son 1, `calculation` es igual a 2 + 2 - 3 = 1, y el resultado es 1. Si solo una de las entradas es 1, `calculation` es igual a 2 + 0 - 3 = -1, y el resultado es 0. Pero, si ambas entradas son 0, `calculation` es igual a -3 y el resultado es 0.

Utilizando un razonamiento parecido, podríamos crear una puerta OR con este código:

```
or_weights = [2., 2]
or_bias = -1.
assert perceptron_output(or_weights, or_bias, [1, 1]) == 1
assert perceptron_output(or_weights, or_bias, [0, 1]) == 1
assert perceptron_output(or_weights, or_bias, [1, 0]) == 1
assert perceptron_output(or_weights, or_bias, [0, 0]) == 0
```

También podríamos crear una puerta NOT (que tiene una sola entrada y convierte 1 en 0 y 0 en 1) con:

```
not_weights = [-2.]
not_bias = 1.
assert perceptron_output(not_weights, not_bias, [0]) == 1
assert perceptron_output(not_weights, not_bias, [1]) == 0
```

Sin embargo, hay algunos problemas que simplemente no se pueden resolver con un solo perceptrón. Por ejemplo, por mucho que se intente, no se puede usar un perceptrón para crear una puerta XOR que dé como resultado 1 si exactamente una de sus entradas es 1 y 0 si no lo es. Aquí es donde empezamos a necesitar redes neuronales más complicadas.

Por supuesto, no hace falta aproximarse a una neurona para poder crear una puerta lógica:

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

Como ocurre con las neuronas de verdad, las artificiales empiezan a resultar más interesantes cuando se las empieza a conectar unas con otras.

Redes neuronales prealimentadas

La topología del cerebro es enormemente complicada, de ahí que sea habitual aproximarse a ella con una red neuronal prealimentada teórica, formada por capas discretas de neuronas, cada una de ellas conectada con la siguiente. Normalmente esto conlleva una capa de entrada (que recibe entradas y las transmite sin cambios), una o varias "capas ocultas" (cada una de las cuales consiste en neuronas que toman las salidas de la capa anterior, realizan algún tipo de cálculo y pasan el resultado a la siguiente capa) y una capa de salida (que produce los resultados finales).

Exactamente igual que en el perceptrón, cada neurona (no de entrada) tiene un peso correspondiente a cada una de sus entradas y un sesgo. Para que nuestra representación sea más sencilla, añadiremos el sesgo al final de nuestro vector de pesos y daremos a cada neurona una entrada de sesgo que siempre es igual a 1.

Igual que con el perceptrón, para cada neurona sumaremos los productos de sus entradas y sus pesos. Pero aquí, en lugar de dar como resultado `step_function` aplicado a dicho producto, obtendremos una aproximación suave de él. Lo que emplearemos es la función `sigmoid` (figura 18.2):

```
import math
def sigmoid(t: float) -> float:
   return 1 / (1 + math.exp(-t))
```

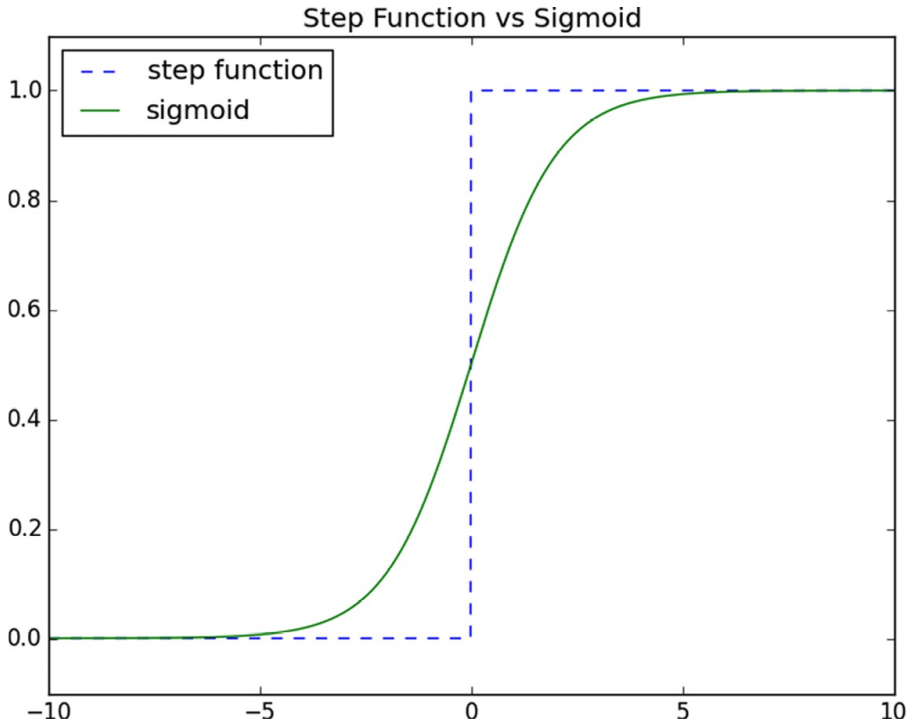


Figura 18.2. La función sigmoid.

¿Por qué utilizar `sigmoid` en lugar de la más sencilla `step_function`? Para entrenar una red neuronal hay que utilizar el cálculo, y para ello se necesitan funciones suaves. `step_function` ni siquiera es continua, y `sigmoid` es una buena aproximación suave de ella.

Nota: Quizá recuerde `sigmoid` del capítulo 16, donde se llamaba `logistic`. Técnicamente, "sigmoide" se refiere a la forma de la función y "logístico" a su función específica, aunque la gente suele utilizar ambos términos de forma intercambiable.

Calculamos entonces el resultado como:

```
def neuron_output(weights: Vector, inputs: Vector) -> float:
   # weights incluye el término de sesgo, inputs incluye un 1
   return sigmoid(dot(weights, inputs))
```

Dada esta función, podemos representar una neurona simplemente como un vector de pesos cuya longitud es una más que el número de entradas a esa

neurona (debido al peso del sesgo). Entonces podemos representar una red neuronal como una lista de capas (no de entrada), donde cada capa no es más que una lista de las neuronas de dicha capa.

Es decir, representaremos una red neuronal como una lista (capas) de listas (neuronas) de vectores (pesos).

Dada una representación como esta, utilizar la red neuronal es bastante sencillo:

```
from typing import List
def feed_forward(neural_network: List[List[Vector]],
                 input_vector: Vector) -> List[Vector]:
    """
    Feeds the input vector through the neural network.
    Returns the outputs of all layers (not just the last one).
    """
    outputs: List[Vector] = []
    for layer in neural_network:
        input_with_bias = input_vector + [1]              # Suma una constante.
        output = [neuron_output(neuron,                   # Calcula la entrada
        input_with_bias)
                  for neuron in layer]                    # para cada neurona.
        outputs.append(output)                            # Suma a los
                                                          resultados.
        # Luego la entrada de la capa siguiente es la salida de esta
        input_vector = output
    return outputs
```

Ahora es fácil crear la puerta XOR que no pudimos construir con un solo perceptrón. Solo tenemos que dimensionar los pesos hacia arriba de forma que las funciones `neuron_output` estén o bien realmente cerca de 0 o realmente cerca de 1:

```
xor_network = [# capa oculta
               [[20., 20, -30],        # neurona 'and'
                  [20., 20, -10]],     # neurona 'or'
               # output layer
               [[-60., 60, -30]]]      # neurona '2ª entrada pero no 1ª entrada'
# feed_forward devuelve las salidas de todas las capas, por lo que el [-1]
# obtiene la salida final, y el [0] saca el valor del vector resultante
assert 0.000 < feed_forward(xor_network, [0, 0])[-1][0] < 0.001
assert 0.999 < feed_forward(xor_network, [1, 0])[-1][0] < 1.000
assert 0.999 < feed_forward(xor_network, [0, 1])[-1][0] < 1.000
```

```
assert 0.000 < feed_forward(xor_network, [1, 1])[-1][0] < 0.001
```

Para una determinada entrada (que es un vector bidimensional), la capa oculta produce un vector bidimensional que consiste en el “and” de los dos valores de entrada y el “or” de los dos valores de entrada.

Y la capa de salida toma un vector bidimensional y calcula “segundo elemento pero no primer elemento”. El resultado es una red que efectúa “or, pero no and”, que es justamente XOR (figura 18.3).

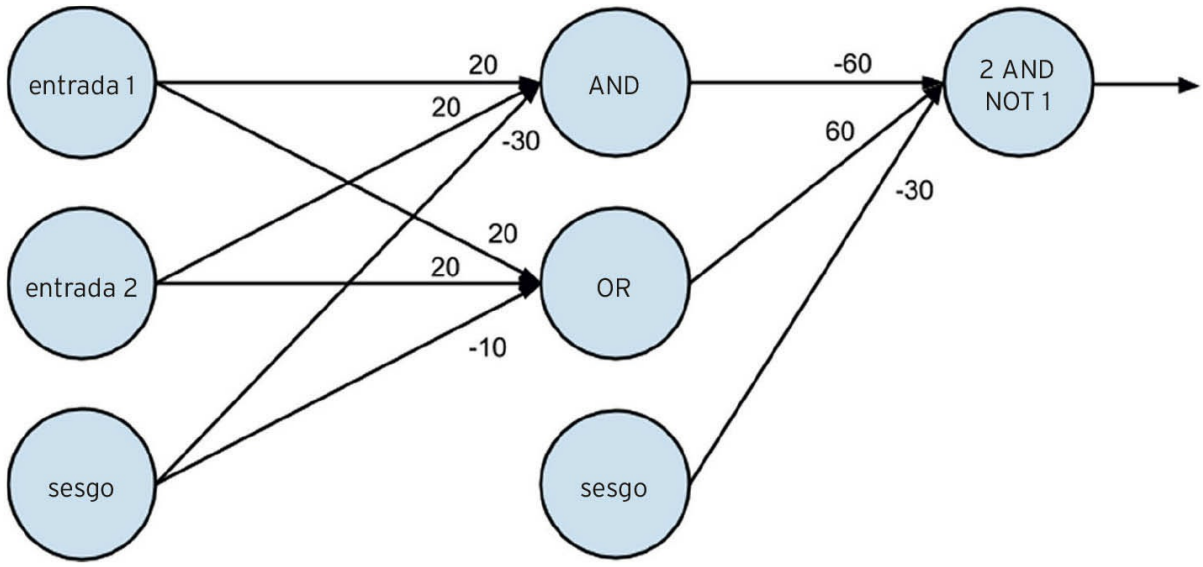


Figura 18.3. Una red neuronal para XOR.

Una sugerencia para pensar en esto es que la capa oculta está calculando características de los datos de entrada (en este caso “and” y “or”) y la capa de salida está combinando esas características de una forma que genere el resultado deseado.

Retropropagación

Normalmente no se crean redes neuronales a mano, en parte porque se utilizan para resolver problemas mucho mayores (un problema de reconocimiento de imagen implicaría cientos o miles de neuronas), y en parte

porque normalmente no somos capaces de “razonar” lo que son las neuronas. En vez de ello (como es habitual), empleamos datos para entrenar redes neuronales. El enfoque normal es un algoritmo denominado retropropagación, que utiliza descenso de gradiente o una de sus variantes.

Supongamos que tenemos un conjunto de entrenamiento que consiste en vectores de entrada y vectores de salida objetivo correspondientes. Por ejemplo, en nuestro anterior ejemplo `xor_network`, el vector de entrada `[1, 0]` correspondía a la salida objetivo [1]. Imaginemos que nuestra red tiene un cierto conjunto de pesos. Los ajustamos entonces utilizando el siguiente algoritmo:

1. Ejecuta `feed_forward` en un vector de entrada para producir las salidas de todas las neuronas de la red.
2. Conocemos la salida objetivo, de forma que calculamos una pérdida que es la suma de los errores cuadrados.
3. Calcula el gradiente de esta pérdida como una función de los pesos de la neurona de salida.
4. “Propaga” los gradientes y errores hacia atrás para calcular los gradientes con respecto a los pesos de las neuronas ocultas.
5. Da un paso de descenso de gradiente.

Normalmente, ejecutamos este algoritmo muchas veces para todo nuestro conjunto de entrenamiento hasta que la red converge.

Para empezar, escribamos la función para calcular los gradientes:

```
def sqerror_gradients(network: List[List[Vector]],
                      input_vector: Vector,
                      target_vector: Vector) -> List[List[Vector]]:
    """
    Given a neural network, an input vector, and a target vector,
    make a prediction and compute the gradient of the squared error
    loss with respect to the neuron weights.
    """
    # paso adelante
    hidden_outputs, outputs = feed_forward(network, input_vector)
    # gradientes con respecto a obtener salidas de preactivación de neurona
    output_deltas = [output * (1–output) * (output–target)
                     for output, target in zip(outputs, target_vector)]
```

```
  # gradientes con respecto a obtener salidas de pesos de neurona
  output_grads = [[output_deltas[i] * hidden_output
for hidden_output in hidden_outputs + [1]]
            for i, output_neuron in enumerate(network[-1])]
  # gradientes con respecto a salidas ocultas de preactivación de neurona
  hidden_deltas = [hidden_output * (1–hidden_output) *
              dot(output_deltas, [n[i] for n in network[-1]])
            for i, hidden_output in enumerate(hidden_outputs)]
  # gradientes con respecto a salidas ocultas de pesos de neurona
  hidden_grads = [[hidden_deltas[i] * input for input in input_vector + [1]]
            for i, hidden_neuron in enumerate(network[0])]
return [hidden_grads, output_grads]
```

Las matemáticas en que se basan los cálculos anteriores no son muy difíciles, pero implican aplicar un poco de cálculo tedioso y una minuciosa atención al detalle, de modo que se queda como ejercicio para el lector.

Armados con la capacidad de calcular gradientes, ahora podemos entrenar redes neuronales. Intentemos descubrir la red XOR que diseñamos antes a mano.

Empezaremos generando los datos de entrenamiento e inicializando nuestra red neuronal con pesos aleatorios:

```
import random
random.seed(0)
# datos de entrenamiento
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]]
# empieza con pesos aleatorios
network =  # capa oculta : 2 entradas -> 2 salidas
[
          [[random.random() for _ in range(2 + 1)],      # 1ª neurona oculta
           [random.random() for _ in range(2 +           # 2ª neurona oculta
           1)]],
          # capa de salida : 2 entradas -> 1 salida
          [[random.random() for _ in range(2 + 1)]]      # 1ª neurona de
                                                         salida
       ]
```

Como es habitual, podemos entrenarla utilizando descenso de gradiente. Una diferencia con respecto a nuestros ejemplos anteriores es que aquí tenemos varios vectores de parámetro, cada uno con su propio gradiente, lo

que significa que tendremos que llamar a `gradient_step` en cada uno de ellos.

```
from scratch.gradient_descent import gradient_step
import tqdm
learning_rate = 1.0
for epoch in tqdm.trange(20000, desc="neural net for xor"):
   for x, y in zip(xs, ys):
      gradients = sqerror_gradients(network, x, y)
      # Da un paso de gradiente para cada neurona en cada capa
      network = [[gradient_step(neuron, grad, -learning_rate)
 for neuron, grad in zip(layer, layer_grad)]
              for layer, layer_grad in zip(network, gradients)]
# comprueba que descubrió XOR
assert feed_forward(network, [0, 0])[-1][0] < 0.01
assert feed_forward(network, [0, 1])[-1][0] > 0.99
assert feed_forward(network, [1, 0])[-1][0] > 0.99
assert feed_forward(network, [1, 1])[-1][0] < 0.01
```

Para mí, la red resultante tiene pesos que tienen este aspecto:

```
[ # capa oculta
   [[7, 7, -3],         # calcula OR
  [5, 5, -8]],          # calcula AND
   # capa de salida
   [[11, -12, -5]]      # calcula "primero pero no segundo"
]
```

Que es bastante similar en concepto a nuestra anterior red creada a medida.

Ejemplo: Fizz Buzz

El vicepresidente de Ingeniería quiere entrevistar a candidatos técnicos haciéndoles resolver el test de “Fizz Buzz”, el siguiente desafío de programación muy utilizado:

```
Mostrar en pantalla los números de 1 a 100, pero si el número es divisible
por 3, mostrar "fizz"; si el número es divisible por 5, mostrar "buzz",
y si el número es divisible por 15, mostrar "fizzbuzz".
```

El vicepresidente cree que la capacidad para resolver esto demuestra una extrema habilidad para programar. Pero usted piensa que este problema es tan fácil que una red neuronal podría resolverlo.

Las redes neuronales toman vectores como entradas y producen vectores como salidas. Como ya hemos dicho, el problema de programación es convertir un entero en una cadena de texto. Así que el primer desafío es encontrar una forma de reformularlo como un problema de vectores.

Para las salidas no es complicado: hay básicamente cuatro clases de salidas, de modo que podemos codificar la salida como un vector de cuatro ceros y unos:

```
def fizz_buzz_encode(x: int) -> Vector:
   if x % 15 == 0:
      return [0, 0, 0, 1]
   elif x % 5 == 0:
      return [0, 0, 1, 0]
   elif x % 3 == 0:
      return [0, 1, 0, 0]
   else:
      return [1, 0, 0, 0]
assert fizz_buzz_encode(2) == [1, 0, 0, 0]
assert fizz_buzz_encode(6) == [0, 1, 0, 0]
assert fizz_buzz_encode(10) == [0, 0, 1, 0]
assert fizz_buzz_encode(30) == [0, 0, 0, 1]
```

Utilizaremos esto para generar nuestros vectores objetivo. Los vectores de entrada son menos obvios. No queremos utilizar solamente un vector unidimensional que contenga el número de entrada por un par de razones. Una sola entrada captura una “intensidad”, pero el hecho de que 2 sea el doble de 1 y 4 sea a su vez el doble no parece relevante para este problema. Además, solamente con una entrada la capa oculta no sería capaz de calcular características muy interesantes, lo que significa que probablemente no lograría resolver el problema.

Resulta que una cosa que funciona razonablemente bien es convertir cada número a su representación binaria de unos y ceros (no se preocupe, esto no es obvio; al menos no lo era para mí).

```
def binary_encode(x: int) -> Vector:
```

```
    binary: List[float] = []
    for i in range(10):
        binary.append(x % 2)
        x = x // 2
    return binary
#                           1 2 4 8 16 32 64 128 256 512
assert binary_encode(0)   == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
assert binary_encode(1)   == [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
assert binary_encode(10)  == [0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
assert binary_encode(101) == [1, 0, 1, 0, 0, 1, 1, 0, 0, 0]
assert binary_encode(999) == [1, 1, 1, 0, 0, 1, 1, 1, 1, 1]
```

Como el objetivo es construir las salidas para los números 1 a 100, entrenar según esos números sería hacer trampa. Por lo tanto, entrenaremos según los números 101 a 1.023 (que es el número más grande que podemos representar con 10 dígitos binarios):

```
xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]
```

A continuación, crearemos una red neuronal con pesos iniciales aleatorios. Tendremos 10 neuronas de entrada (ya que estamos representando nuestras entradas como vectores de diez dimensiones) y 4 neuronas de salida (ya que estamos representando nuestros objetivos como vectores de cuatro dimensiones). Le daremos 25 unidades ocultas, pero emplearemos una variable para que sea fácil de cambiar:

```
NUM_HIDDEN = 25
network = [
    # capa oculta : 10 entradas -> NUM_HIDDEN salidas
    [[random.random() for _ in range(10 + 1)] for _ in range(NUM_HIDDEN)],
    # output_layer: NUM_HIDDEN entradas -> 4 salidas
    [[random.random() for _ in range(NUM_HIDDEN + 1)] for _ in range(4)]
]
```

Eso es. Ahora estamos listos para entrenar. Como es un problema más complicado (y hay muchas más cosas que se podrían estropear), probablemente querremos controlar de cerca el proceso de entrenamiento. En particular, para cada *epoch* controlaremos la suma de errores cuadrados y la mostraremos en pantalla. Queremos asegurarnos de que disminuyen:

```
from scratch.linear_algebra import squared_distance
learning_rate = 1.0
with tqdm.trange(500) as t:
   for epoch in t:
      epoch_loss = 0.0
      for x, y in zip(xs, ys):
         predicted = feed_forward(network, x)[-1]
         epoch_loss += squared_distance(predicted, y)
         gradients = sqerror_gradients(network, x, y)
         # Da un paso de gradiente para cada neurona en cada capa
         network = [[gradient_step(neuron, grad, -learning_rate)
                 for neuron, grad in zip(layer, layer_grad)]
              for layer, layer_grad in zip(network, gradients)]
   t.set_description(f"fizz buzz (loss: {epoch_loss:.2f})")
```

Se tarda un poco en entrenar esto, pero al final la pérdida debería empezar a tocar fondo.

Por fin estamos listos para resolver nuestro problema inicial. Solamente nos queda una cosa. Nuestra red producirá un vector de números de cuatro dimensiones, pero queremos una única predicción. Lo haremos tomando `argmax`, que es el índice del valor más grande:

```
def argmax(xs: list) -> int:
   """Returns the index of the largest value"""
   return max(range(len(xs)), key=lambda i: xs[i])
assert argmax([0, -1]) == 0                # items[0] es el más grande
assert argmax([-1, 0]) == 1                # items[1] es el más grande
assert argmax([-1, 10, 5, 20, -3]) == 3    # items[3] es el más grande
```

Ahora podemos resolver por fin “FizzBuzz”:

```
num_correct = 0
for n in range(1, 101):
   x = binary_encode(n)
   predicted = argmax(feed_forward(network, x)[-1])
   actual = argmax(fizz_buzz_encode(n))
   labels = [str(n), "fizz", "buzz", "fizzbuzz"]
   print(n, labels[predicted], labels[actual])
   if predicted == actual:
      num_correct += 1
print(num_correct, "/", 100)
```

Para mí, la red entrenada obtiene 96 de 100 aciertos, que es bastante más del umbral de contratación del vicepresidente de Ingeniería. Ante la evidencia, cede y cambia el reto de la entrevista por “Invertir un árbol binario”.

Para saber más

- Siga leyendo: el capítulo 19 explorará estos temas con mucho más detalle.
- Mi post “Fizz Buzz in Tensorflow”, en `https://joelgrus.com/2016/05/23/fizz-buzz-in-tensorflow/`, es bastante bueno.

19 Deep learning (aprendizaje profundo)

Un poco de aprendizaje es algo peligroso; bebe mucho, o ni pruebes el manantial de Pieria.

—Alexander Pope

El término *deep learning* o aprendizaje profundo hacía originalmente referencia a la aplicación de redes neuronales "profundas" (es decir, redes con más de una capa oculta), aunque en la práctica el término se ajusta en la actualidad a una amplia variedad de arquitecturas neuronales (incluyendo las "sencillas" redes neuronales que desarrollamos en el capítulo 18).

En este capítulo continuaremos nuestro trabajo anterior y veremos una mayor variedad de este tipo de redes y, para ello, introduciremos una serie de abstracciones que nos permitan pensar en ellas de una forma más general.

El tensor

Anteriormente distinguimos entre vectores (*arrays* unidimensionales) y matrices (*arrays* bidimensionales). Cuando empecemos a trabajar con redes más complicadas, necesitaremos utilizar también *arrays* multidimensionales.

En muchas librerías de redes neuronales, a los *arrays* de *n* dimensiones se les denomina tensores, que es como también les llamaremos nosotros (existen pretenciosas razones matemáticas para no llamar así a los tensores; si usted es uno de los sabiondos que opina así, queda anotada su objeción).

Si estuviéramos escribiendo un libro completo sobre *deep learning*, implementaríamos una clase `Tensor` entera que sobrecargara las operaciones aritméticas de Python y pudiera gestionar otras variadas operaciones. Pero una implementación así requeriría un capítulo entero para ella sola. Aquí

vamos a hacer trampa y diremos que una clase `Tensor` es simplemente una `list`. Esto es cierto en un sentido (todos nuestros vectores, matrices y similares de muchas dimensiones son listas), pero sin duda no lo es en el otro (la mayoría de las listas de Python no son *arrays* de *n* dimensiones del modo que a nosotros nos interesa).

Nota: Lo ideal es que hiciéramos algo como esto:

```
# Una Tensor es un float o una List de Tensors
Tensor = Union[float, List[Tensor]]
```

Sin embargo, Python no permitirá definir tipos recursivos así. E incluso aunque lo permitiera, dicha definición sigue sin ser correcta, ya que permite "tensores" erróneos como:

```
[[1.0, 2.0],
[3.0]]
```

Cuyas filas tienen distintos tamaños, lo que hace que no sea un *array* de *n* dimensiones.

Así, como dije antes, simplemente haremos trampa:

```
Tensor = list
```

Y escribiremos una función auxiliar para averiguar la forma de un tensor:

```
from typing import List
def shape(tensor: Tensor) -> List[int]:
   sizes: List[int] = []
   while isinstance(tensor, list):
      sizes.append(len(tensor))
      tensor = tensor[0]
   return sizes
assert shape([1, 2, 3]) == [3]
assert shape([[1, 2], [3, 4], [5, 6]]) == [3, 2]
```

Como los tensores pueden tener cualquier cantidad de dimensiones, normalmente tendremos que trabajar con ellos de forma recursiva. Haremos una cosa en el caso unidimensional y lo repetiremos recursivamente el caso multidimensional:

```
def is_1d(tensor: Tensor) -> bool:
   """
   If tensor[0] is a list, it's a higher-order tensor.
   Otherwise, tensor is 1-dimensional (that is, a vector).
   """
   return not isinstance(tensor[0], list)
assert is_1d([1, 2, 3])
assert not is_1d([[1, 2], [3, 4]])
```

Que podemos usar para escribir una función recursiva `tensor_sum`:

```
def tensor_sum(tensor: Tensor) -> float:
   """Sums up all the values in the tensor"""
   if is_1d(tensor):
      return sum(tensor)               # solo una lista de float, usa suma de
                                       Python
   else:
      return                           # Llama a tensor_sum en cada fila
      sum(tensor_sum(tensor_i)
              for tensor_i in          # y suma esos resultados.
              tensor)
assert tensor_sum([1, 2, 3]) == 6
assert tensor_sum([[1, 2], [3, 4]]) == 10
```

Si no se está acostumbrado a pensar recursivamente, conviene meditarlo hasta que tenga sentido, porque utilizaremos la misma lógica en todo este capítulo. No obstante, crearemos un par de funciones auxiliares, de manera que no tengamos que reescribir esta lógica continuamente. La primera aplica una función basada en elementos a un solo tensor:

```
from typing import Callable
def tensor_apply(f: Callable[[float], float], tensor: Tensor) -> Tensor:
   """Applies f elementwise"""
   if is_1d(tensor):
      return [f(x) for x in tensor]
   else:
      return [tensor_apply(f, tensor_i) for tensor_i in tensor]
assert tensor_apply(lambda x: x + 1, [1, 2, 3]) == [2, 3, 4]
assert tensor_apply(lambda x: 2 * x, [[1, 2], [3, 4]]) == [[2, 4], [6, 8]]
```

Podemos utilizarla para escribir otra función que crea un tensor nulo con la misma forma que un cierto tensor dado:

```
def zeros_like(tensor: Tensor) -> Tensor:
   return tensor_apply(lambda _: 0.0, tensor)
assert zeros_like([1, 2, 3]) == [0, 0, 0]
assert zeros_like([[1, 2], [3, 4]]) == [[0, 0], [0, 0]]
```

También tendremos que aplicar una función a los elementos correspondientes de dos tensores (que más vale que tengan exactamente la misma forma, aunque no vamos a comprobarlo):

```
def tensor_combine(f: Callable[[float, float], float],
             t1: Tensor,
             t2: Tensor) -> Tensor:
   """Applies f to corresponding elements of t1 and t2"""
   if is_1d(t1):
      return [f(x, y) for x, y in zip(t1, t2)]
   else:
      return [tensor_combine(f, t1_i, t2_i)
           for t1_i, t2_i in zip(t1, t2)]
```

```
import operator
```

```
assert tensor_combine(operator.add, [1, 2, 3], [4, 5, 6]) == [5, 7, 9]
```

```
assert tensor_combine(operator.mul, [1, 2, 3], [4, 5, 6]) == [4, 10, 18]
```

La capa de abstracción

En el capítulo anterior, creamos una sencilla red neuronal que nos permitía apilar dos capas de neuronas, cada una de las cuales realizaba el cálculo `sigmoid(dot(weighs, inputs))`.

Aunque esta sea quizá una representación idealizada de lo que hace realmente una neurona, en la práctica querríamos permitir una mayor variedad de tareas. Quizá nos gustaría que las neuronas recordaran algo sobre sus anteriores entradas, o quizá preferiríamos utilizar una función de activación que no fuera `sigmoid`. Con frecuencia, estaría bien emplear más de dos capas (en realidad, nuestra función `feed_forward` gestionaba cualquier

número de capas, pero nuestros cálculos de gradiente no).

En este capítulo, crearemos mecanismos para implementar tal variedad de redes neuronales. Nuestra abstracción fundamental será `Layer`, que sabe cómo aplicar una cierta función a sus entradas y cómo retropropagar gradientes.

Una forma de pensar en las redes neuronales creadas en el capítulo 18 es como si fueran una capa “lineal” seguida de una capa “sigmoide”, después otra capa lineal y otra sigmoide. No las distinguíamos en estos términos, pero hacerlo nos permitirá experimentar con estructuras mucho más generales:

```
from typing import Iterable, Tuple
class Layer:
   """
   Our neural networks will be composed of Layers, each of which
   knows how to do some computation on its inputs in the "forward"
   direction and propagate gradients in the "backward" direction.
   """
   def forward(self, input):
      """
      Note the lack of types. We’re not going to be prescriptive
      about what kinds of inputs layers can take and what kinds
      of outputs they can return.
      """
      raise NotImplementedError
   def backward(self, gradient):
      """
      Similarly, we’re not going to be prescriptive about what the
      gradient looks like. It’s up to you the user to make sure
      that you’re doing things sensibly.
      """
      raise NotImplementedError
   def params(self) -> Iterable[Tensor]:
      """
      Returns the parameters of this layer. The default implementation
      returns nothing, so that if you have a layer with no parameters
      you don’t have to implement this.
      """
      return ()
   def grads(self) -> Iterable[Tensor]:
      """
      Returns the gradients, in the same order as params().
      """
      return ()
```

Habrá que implementar los métodos `forward` y `backward` en nuestras subclases concretas. Una vez creada una red neuronal, querremos entrenarla utilizando descenso de gradiente, lo que significa que tendremos que actualizar cada parámetro de la red mediante su gradiente. Por consiguiente, insistimos en que cada capa sea capaz de indicarnos sus parámetros y gradientes.

Algunas capas (por ejemplo, una que aplica `sigmoid` a cada una de sus entradas) no tienen parámetros que actualizar, de modo que proporcionamos una implementación predeterminada que gestione esa situación. Veamos esa capa:

```
from scratch.neural_networks import sigmoid
class Sigmoid(Layer):
   def forward(self, input: Tensor) -> Tensor:
      """
      Apply sigmoid to each element of the input tensor,
      and save the results to use in backpropagation.
      """
      self.sigmoids = tensor_apply(sigmoid, input)
      return self.sigmoids
   def backward(self, gradient: Tensor) -> Tensor:
      return tensor_combine(lambda sig, grad: sig * (1–sig) * grad,
self.sigmoids,
gradient)
```

Aquí hay un par de cosas a tener en cuenta. Una es que, durante el paso adelante, guardamos los sigmoides calculados para poder utilizarlos más tarde en el paso atrás. Nuestras capas tendrán que hacer habitualmente este tipo de cosas.

Segunda, quizá se esté preguntando de dónde viene `sig * (1 - sig) * grad`. No es más que la regla de la cadena de cálculo y corresponde al término `output * (1 - output) * (output - target)` de nuestras anteriores redes neuronales.

Por último, se puede ver cómo pudimos hacer uso de las funciones `tensor_apply` y `tensor_combine`. La mayor parte de nuestras capas utilizarán estas funciones de forma similar.

La capa lineal

La otra pieza que nos hará falta para duplicar las redes neuronales del capítulo 18 es una capa "lineal", que represente la parte `dot(weights, inputs)` de las neuronas. Esta capa tendrá parámetros, que nos gustaría inicializar con valores aleatorios.

Resulta que los valores iniciales de los parámetros pueden marcar una gran diferencia en lo rápido que entrena la red (y a veces en si entrena). Si los pesos son demasiado grandes, pueden producir salidas grandes en un rango en el que la función de activación tiene gradientes que son casi cero. Pero las partes de la red que tienen gradientes nulos no pueden necesariamente aprender nada mediante el descenso de gradiente.

En consecuencia, implementaremos tres esquemas distintos para generar aleatoriamente nuestros tensores de peso. El primero es elegir cada valor desde la distribución uniforme aleatoria en [0, 1] (es decir, como `random.random()`). El segundo (y predeterminado) es seleccionar cada valor aleatoriamente desde una distribución normal estándar. Y el tercero es utilizar la inicialización de Xavier, donde cada peso se inicializa aleatoriamente desde una distribución normal con media 0 y varianza 2 / (`num_inputs + num_outputs`). Resulta que esto suele funcionar muy bien con pesos de redes neuronales. Los implementaremos con una función `random_uniform` y otra `random_normal`:

```
import random
from scratch.probability import inverse_normal_cdf
def random_uniform(*dims: int) -> Tensor:
   if len(dims) == 1:
      return [random.random() for _ in range(dims[0])]
   else:
      return [random_uniform(*dims[1:]) for _ in range(dims[0])]
   def random_normal(*dims: int,
                mean: float = 0.0,
                variance: float = 1.0) -> Tensor:
      if len(dims) == 1:
         return [mean + variance * inverse_normal_cdf(random.random())
              for _ in range(dims[0])]
      else:
```

```
        return [random_normal(*dims[1:], mean=mean, variance=variance)
             for _ in range(dims[0])]
   assert shape(random_uniform(2, 3, 4)) == [2, 3, 4]
   assert shape(random_normal(5, 6, mean=10)) == [5, 6]
```

Después, los envolvemos todos en una función `random_tensor`:

```
def random_tensor(*dims: int, init: str = 'normal') -> Tensor:
   if init == 'normal':
      return random_normal(*dims)
   elif init == 'uniform':
      return random_uniform(*dims)
   elif init == 'xavier':
      variance = len(dims) / sum(dims)
      return random_normal(*dims, variance=variance)
   else:
      raise ValueError(f"unknown init: {init}")
```

Ahora podemos definir nuestra capa lineal. Tenemos que inicializarla con la dimensión de las entradas (lo que nos dice cuántos pesos necesita cada neurona), la dimensión de las salidas (que nos indica cuántas neuronas deberíamos tener) y el esquema de inicialización que queremos:

```
from scratch.linear_algebra import dot
class Linear(Layer):
   def __init__(self,
 input_dim: int,
 output_dim: int,
 init: str = 'xavier') -> None:
   """
   A layer of output_dim neurons, each with input_dim weights
   (and a bias).
   """
   self.input_dim = input_dim
   self.output_dim = output_dim
   # self.w[o] es los pesos para la neurona o
   self.w = random_tensor(output_dim, input_dim, init=init)
   # self.b[o] es del término de sesgo para la neurona o
   self.b = random_tensor(output_dim, init=init)
```

Nota: En el caso de que se esté preguntando cuán importantes son los esquemas de inicialización, me fue imposible entrenar algunas de las redes de

este capítulo con inicializaciones diferentes a las que utilicé.

El método `forward` es fácil de implementar. Obtendremos una salida por neurona, que meteremos en un vector. La salida de cada neurona no es más que el `dot` de sus pesos con la entrada, más su sesgo:

```
def forward(self, input: Tensor) -> Tensor:
   # Guarda la entrada para usarla en el paso atrás.
   self.input = input
   # Devuelve el vector de salidas de la neurona.
   return [dot(input, self.w[o]) + self.b[o]
        for o in range(self.output_dim)]
```

El método `backward` es más complicado, pero sabiendo cálculo no es difícil:

```
def backward(self, gradient: Tensor) -> Tensor:
# Cada b[o] es sumado a output[o], con lo que
# el gradiente de b es el mismo que el gradiente de salida.
self.b_grad = gradient
# Cada w[o][i] multiplica input[i] y es sumado a output[o].
# Así su gradiente es input[i] * gradient[o].
self.w_grad = [[self.input[i] * gradient[o]
 for i in range(self.input_dim)]
           for o in range(self.output_dim)]
# Cada input[i] multiplica cada w[o][i] y es sumado a cada
# output[o]. Así su gradiente es la suma de w[o][i] * gradient[o]
# en todas las salidas.
return [sum(self.w[o][i] * gradient[o] for o in range(self.output_dim))
     for i in range(self.input_dim)]
```

Nota: En una librería de tensores “real”, estas operaciones (y muchas otras) se representarían como multiplicaciones de matriz o tensor, operaciones que estas librerías están diseñadas para hacer muy rápidamente. La nuestra es muy lenta.

Lo último que necesitamos hacer aquí es implementar `params` y `grads`. Tenemos dos parámetros y dos gradientes asociados:

```
  def params(self) -> Iterable[Tensor]:
    return [self.w, self.b]
  def grads(self) -> Iterable[Tensor]:
    return [self.w_grad, self.b_grad]
```

Redes neuronales como una secuencia de capas

Nos gustaría pensar en las redes neuronales como secuencias de capas, así que vamos a idear una forma de combinar varias capas en una. La red neuronal resultante es en sí misma una capa, e implementa los métodos `Layer` de las maneras obvias:

```
from typing import List
class Sequential(Layer):
  """
  A layer consisting of a sequence of other layers.
  It's up to you to make sure that the output of each layer
  makes sense as the input to the next layer.
  """
  def __init__(self, layers: List[Layer]) -> None:
    self.layers = layers
  def forward(self, input):
    """Just forward the input through the layers in order."""
    for layer in self.layers:
      input = layer.forward(input)
    return input
  def backward(self, gradient):
    """Just backpropagate the gradient through the layers in reverse."""
    for layer in reversed(self.layers):
      gradient = layer.backward(gradient)
    return gradient
  def params(self) -> Iterable[Tensor]:
    """Just return the params from each layer."""
    return (param for layer in self.layers for param in layer.params())
  def grads(self) -> Iterable[Tensor]:
    """Just return the grads from each layer."""
    return (grad for layer in self.layers for grad in layer.grads())
```

Así, podríamos representar la red neuronal que hemos utilizado para XOR como:

```
xor_net = Sequential([
   Linear(input_dim=2, output_dim=2),
   Sigmoid(),
   Linear(input_dim=2, output_dim=1),
   Sigmoid()
])
```

Pero aún necesitamos más instrumentos para entrenarla.

Pérdida y optimización

Anteriormente, escribimos para nuestros modelos funciones de pérdida y gradiente individuales. Ahora queremos experimentar con distintas funciones de pérdida, de modo que (como es habitual) introduciremos una nueva abstracción `Loss` que encapsula tanto el cálculo de pérdida como el de gradiente:

```
class Loss:
   def loss(self, predicted: Tensor, actual: Tensor) -> float:
      """How good are our predictions? (Larger numbers are worse.)"""
      raise NotImplementedError
   def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
      """How does the loss change as the predictions change?"""
      raise NotImplementedError
```

Ya hemos trabajado muchas veces con la pérdida que equivale a la suma de los errores cuadrados, así que debería resultarnos sencillo implementarla. El único añadido es que tendremos que utilizar `tensor_combine`:

```
class SSE(Loss):
   """Loss function that computes the sum of the squared errors."""
   def loss(self, predicted: Tensor, actual: Tensor) -> float:
      # Calcula el tensor de diferencias al cuadrado
      squared_errors = tensor_combine(
         lambda predicted, actual: (predicted-actual) ** 2,
         predicted,
         actual)
      # Y simplemente los suma
      return tensor_sum(squared_errors)
```

```
  def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
    return tensor_combine(
      lambda predicted, actual: 2 * (predicted–actual),
      predicted,
      actual)
```

(Veremos en un momento una función de pérdida diferente).

La última pieza que falta es el descenso de gradiente. A lo largo del libro hemos hecho manualmente todo el descenso de gradiente teniendo un bucle de entrenamiento que conlleva algo como:

```
theta = gradient_step(theta, grad, -learning_rate)
```

En nuestro caso, esto no funcionará por un par de razones. La primera es que nuestras redes neuronales tendrán muchos parámetros, y tendremos que actualizarlos todos. La segunda es que nos gustaría poder utilizar variantes más inteligentes del descenso de gradiente, y no queremos tener que reescribirlas cada vez.

En consecuencia (lo ha adivinado), introduciremos una abstracción `Optimizer`, de la que el descenso de gradiente será una instancia específica:

```
class Optimizer:
  """
  An optimizer updates the weights of a layer (in place) using information
  known by either the layer or the optimizer (or by both).
  """
  def step(self, layer: Layer) -> None:
    raise NotImplementedError
```

Después de esto, es fácil implementar el descenso de gradiente, empleando de nuevo `tensor_combine`:

```
class GradientDescent(Optimizer):
  def __init__(self, learning_rate: float = 0.1) -> None:
    self.lr = learning_rate
  def step(self, layer: Layer) -> None:
    for param, grad in zip(layer.params(), layer.grads()):
      # Actualiza param usando un paso de gradiente
      param[:] = tensor_combine(
        lambda param, grad: param–grad * self.lr,
        param,
```

```
            grad)
```

Lo único que puede resultar sorprendente es la “asignación de segmentos”, que es un reflejo del hecho de que reasignar una lista no cambia su valor original. Es decir, si no hicimos otra cosa que `param = tensor_combine(. . .)`, estaríamos redefiniendo la variable local `param`, pero no se vería afectado el tensor de parámetro original almacenado en la capa. Sin embargo, si asignamos al segmento o *slice* `[:]`, sí que cambia los valores que están dentro de la lista.

Este es un ejemplo sencillo que lo demuestra:

```
tensor = [[1, 2], [3, 4]]
for row in tensor:
   row = [0, 0]
assert tensor == [[1, 2], [3, 4]], "assignment doesn’t update a list"
for row in tensor:
   row[:] = [0, 0]
assert tensor == [[0, 0], [0, 0]], "but slice assignment does"
```

Teniendo poca experiencia con Python, este comportamiento puede resultar sorprendente, de modo que conviene meditar sobre ello y probar ejemplos hasta que tenga sentido.

Para demostrar el valor de esta abstracción, implementemos otro optimizador que utiliza *momentum*. La idea es que no queremos reaccionar en exceso a cada nuevo gradiente, y por eso mantenemos una media activa de los gradientes que hemos visto, actualizándola con cada gradiente nuevo y dando un paso en la dirección de la media:

```
class Momentum(Optimizer):
   def __init__(self,
                learning_rate: float,
                momentum: float = 0.9) -> None:
      self.lr = learning_rate
      self.mo = momentum
      self.updates: List[Tensor] = []              # media activa
   def step(self, layer: Layer) -> None:
      # Si no tenemos actualizaciones previas, empieza con todo ceros
      if not self.updates:
         self.updates = [zeros_like(grad) for grad in layer.grads()]
```

```
      for update, param, grad in zip(self.updates,
                      layer.params(),
                      layer.grads()):
        # Aplica momentum
        update[:] = tensor_combine(
           lambda u, g: self.mo * u + (1-self.mo) * g,
           update,
           grad)
        # Luego da un paso de gradiente
        param[:] = tensor_combine(
           lambda p, u: p-self.lr * u,
           param,
           update)
```

Como hemos utilizado una abstracción `Optimizer`, podemos alternar fácilmente entre nuestros distintos optimizadores.

Ejemplo: XOR revisada

Veamos lo fácil que es usar nuestra nueva estructura para entrenar una red que pueda calcular XOR. Empezamos creando de nuevo los datos de entrenamiento:

```
# datos de entrenamiento
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]]
```

Después, definimos la red, aunque ahora podemos dejar fuera la última capa sigmoide:

```
random.seed(0)
net = Sequential([
   Linear(input_dim=2, output_dim=2),
   Sigmoid(),
   Linear(input_dim=2, output_dim=1)
])
```

Ahora podemos escribir un sencillo bucle de entrenamiento, salvo que en este momento podemos utilizar las abstracciones de `Optimizer` y `Loss`, lo que nos permite probar fácilmente varias:

```
import tqdm
optimizer = GradientDescent(learning_rate=0.1)
loss = SSE()
with tqdm.trange(3000) as t:
   for epoch in t:
      epoch_loss = 0.0
      for x, y in zip(xs, ys):
         predicted = net.forward(x)
         epoch_loss += loss.loss(predicted, y)
         gradient = loss.gradient(predicted, y)
         net.backward(gradient)
         optimizer.step(net)
      t.set_description(f"xor loss {epoch_loss:.3f}")
```

Este código debería entrenar la red rápidamente y, además, debería permitir ver cómo disminuye la pérdida. Ahora podemos inspeccionar los pesos:

```
for param in net.params():
   print(param)
```

Los resultados son, más o menos, para mi red:

```
hidden1 = -2.6 * x1 + -2.7 * x2 + 0.2 # NOR
hidden2 = 2.1 * x1 + 2.1 * x2–3.4 # AND
output = -3.1 * h1 + -2.6 * h2 + 1.8 # NOR
```

Así, `hidden1` se activa si ninguna entrada es 1, `hidden2` se activa si ambas entradas son 1, y `output` se activa si ninguna entrada oculta es 1 (es decir, si no se da el caso de que ninguna entrada sea 1 y tampoco de que ambas entradas sean 1). En realidad, esta es exactamente la lógica de XOR.

Tenga en cuenta que esta red descubrió características diferentes a la que entrenamos en el capítulo 18, pero se las sigue arreglando para hacer lo mismo.

Otras funciones de activación

La función `sigmoid` ha caído en desgracia por dos razones. Una es que

`sigmoid(0)` es igual a 1/2, lo que significa que una neurona cuyas entradas suman 0 tiene una salida positiva. La otra es que su gradiente es muy próximo a 0 para entradas muy grandes y muy pequeñas, lo que significa que sus gradientes pueden "saturarse" y sus pesos pueden quedarse atascados.

Una sustituta habitual es `tanh` ("tangente hiperbólica"), una función distinta con forma de sigmoide que va de -1 a 1 y da como resultado 0 si su entrada es 0. La derivada de `tanh(x)` es sencillamente `1 - tanh(x) ** 2`, con lo que la capa es fácil de escribir:

```
import math
def tanh(x: float) -> float:
   # Si x es muy grande o muy pequeño, tanh es (básicamente) 1 o -1.
   # Comprobamos esto porque, p.ej., math.exp(1000) da un error.
   if x < -100: return -1
   elif x > 100: return 1
   em2x = math.exp(-2 * x)
   return (1–em2x) / (1 + em2x)
class Tanh(Layer):
   def forward(self, input: Tensor) -> Tensor:
      # Guarda la salida de tanh para usar en el paso atrás.
      self.tanh = tensor_apply(tanh, input)
      return self.tanh
   def backward(self, gradient: Tensor) -> Tensor:
      return tensor_combine(
         lambda tanh, grad: (1–tanh ** 2) * grad,
         self.tanh,
         gradient)
```

En redes más grandes otra sustituta habitual es `Relu`, que es 0 para entradas negativas y la identidad para entradas positivas:

```
class Relu(Layer):
   def forward(self, input: Tensor) -> Tensor:
      self.input = input
      return tensor_apply(lambda x: max(x, 0), input)
   def backward(self, gradient: Tensor) -> Tensor:
      return tensor_combine(lambda x, grad: grad if x > 0 else 0,
self.input,
gradient)
```

Hay muchas otras; le animo a que juegue con ellas en sus redes.

Ejemplo: FizzBuzz revisado

Ahora podemos usar nuestra estructura de *deep learning* para reproducir nuestra solución de la sección “Ejemplo: Fizz Buzz” del capítulo 18. Configuremos los datos:

```
from scratch.neural_networks import binary_encode, fizz_buzz_encode, argmax
xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]
```

Ahora creemos la red:

```
NUM_HIDDEN = 25
random.seed(0)
net = Sequential([
   Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
   Tanh(),
   Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform'),
   Sigmoid()
])
```

Como estamos entrenando, controlemos también nuestra precisión en el conjunto de entrenamiento:

```
def fizzbuzz_accuracy(low: int, hi: int, net: Layer) -> float:
   num_correct = 0
   for n in range(low, hi):
      x = binary_encode(n)
      predicted = argmax(net.forward(x))
      actual = argmax(fizz_buzz_encode(n))
      if predicted == actual:
         num_correct += 1
   return num_correct / (hi-low)
optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SSE()
with tqdm.trange(1000) as t:
   for epoch in t:
      epoch_loss = 0.0
      for x, y in zip(xs, ys):
```

```
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)
            optimizer.step(net)
        accuracy = fizzbuzz_accuracy(101, 1024, net)
        t.set_description(f"fb loss: {epoch_loss:.2f} acc: {accuracy:.2f}")
# Ahora comprueba los resultados en el conjunto de prueba
print("test results", fizzbuzz_accuracy(1, 101, net))
```

Tras 1.000 iteraciones de entrenamiento, el modelo obtiene una precisión del 90 % en el conjunto de prueba; si siguiéramos adiestrándolo más tiempo, debería hacerlo incluso mejor (no creo que sea posible entrenar con una precisión del 100 % solo con 25 unidades ocultas, pero sin duda es posible si se aumenta a 50 unidades).

Funciones softmax y entropía cruzada

La red neuronal que utilizamos en la sección anterior terminaba en una capa `Sigmoid`, lo que significa que su salida era un vector de números entre 0 y 1. En particular, podría dar como resultado un vector compuesto enteramente por ceros o por unos. Sin embargo, cuando estamos con problemas de clasificación, nos interesaría obtener un 1 para la clase correcta y un 0 para todas las clases incorrectas. En general, nuestras predicciones no serán tan perfectas, pero al menos nos gustaría predecir una distribución de probabilidad real sobre las clases.

Por ejemplo, si tenemos dos clases, y nuestro modelo da como resultado `[0, 0]`, es difícil encontrarle mucho sentido. ¿No cree el modelo que la salida pertenezca a ninguna de las dos clases?

Pero, si el modelo da como resultado [0.4, 0.6], podemos interpretarlo como una predicción de que hay una probabilidad de 0,4 de que nuestra entrada pertenezca a la primera clase y de 0,6 de que pertenezca a la segunda.

Para conseguir esto, normalmente renunciamos a la capa `Sigmoid` final y utilizamos en su lugar la función `softmax`, que convierte un vector de

números reales en uno de probabilidades. Calculamos `exp(x)` para cada número del vector, lo que da como resultado un vector de números positivos. Después de eso, simplemente dividimos cada uno de esos números positivos por la suma, lo que nos da un montón de números positivos que suman un total de 1 (es decir, un vector de probabilidades).

Si alguna vez terminamos tratando de calcular, digamos, `exp(1000)`, obtendremos un error de Python, de forma que antes de tomar la función `exp` restamos el valor más grande, lo que da como resultado las mismas probabilidades; simplemente es más seguro calcular en Python:

```
def softmax(tensor: Tensor) -> Tensor:
   """Softmax along the last dimension"""
   if is_1d(tensor):
      # Resta el valor más grande por estabilidad numérica.
      largest = max(tensor)
      exps = [math.exp(x–largest) for x in tensor]
      sum_of_exps = sum(exps)          # Este es el "peso" total.
      return [exp_i / sum_of_exps      # La probabilidad es la fracción
           for exp_i in exps]          # del peso total.
   else:
      return [softmax(tensor_i) for tensor_i in tensor]
```

Una vez nuestra red produce probabilidades, a menudo utilizamos otra función de pérdida llamada entropía cruzada (o en ocasiones “probabilidad logarítmica negativa”).

Quizá recuerde que, en la sección “Estimación por máxima verosimilitud” del capítulo 14, justificamos el uso de mínimos cuadrados en la regresión lineal apelando al hecho de que (bajo determinadas suposiciones) los coeficientes de mínimos cuadrados maximizaban la verosimilitud de los datos observados.

Aquí podemos hacer algo parecido: si las salidas de nuestra red son probabilidades, la pérdida de entropía cruzada representa la probabilidad logarítmica negativa de los datos observados, lo que significa que minimizar esa pérdida es lo mismo que maximizar la probabilidad logarítmica (y de ahí la verosimilitud) de los datos de entrenamiento.

Normalmente no incluiríamos la función `softmax` como parte de la red

neuronal como tal, porque resulta que, si `softmax` es parte de nuestra función de pérdida pero no parte de la propia red, los gradientes de la pérdida con respecto a las salidas de la red son muy fáciles de calcular.

```
class SoftmaxCrossEntropy(Loss):
  """
  This is the negative-log-likelihood of the observed values, given the
  neural net model. So if we choose weights to minimize it, our model will
  be maximizing the likelihood of the observed data.
  """
  def loss(self, predicted: Tensor, actual: Tensor) -> float:
    # Aplica softmax para obtener probabilidades
    probabilities = softmax(predicted)
    # Esto será log p_i para la clase real i y 0 para las otras clases.
    # Sumamos una cantidad diminuta a p para evitar tomar log(0).
    likelihoods = tensor_combine(lambda p, act: math.log(p + 1e-30) * act,
probabilities,
actual)
    # Y después simplemente sumamos las negativas.
    return -tensor_sum(likelihoods)
  def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
    probabilities = softmax(predicted)
    # ¿No es esta una ecuación encantadora?
    return tensor_combine(lambda p, actual: p–actual,
probabilities,
actual)
```

Si yo ahora adiestrara la misma red Fizz Buzz utilizando pérdida `SoftmaxCrossEntropy`, descubriría que lo normal es que entrenara mucho más rápido (es decir, en muchos menos *epochs*). Se supone que esto es porque resulta mucho más sencillo encontrar pesos que apliquen `softmax` a una determinada distribución que encontrarlos que apliquen `sigmoid`.

Es decir, si necesito predecir la clase 0 (un vector con un 1 en la primera posición y ceros en las restantes), en el caso `lineal + sigmoid` necesito que la primera salida sea un número positivo grande y las salidas restantes sean números negativos grandes. Pero, en el caso de `softmax`, solo necesito que la primera salida sea más grande que las salidas restantes. Sin duda, hay muchas más formas de que ocurra el segundo caso, lo que sugiere que debería ser más fácil encontrar pesos que lo hicieran realidad:

```
random.seed(0)
net = Sequential([
   Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
   Tanh(),
   Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform')
   # Ahora no hay capa sigmoide final
])
optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SoftmaxCrossEntropy()
with tqdm.trange(100) as t:
   for epoch in t:
      epoch_loss = 0.0
      for x, y in zip(xs, ys):
         predicted = net.forward(x)
         epoch_loss += loss.loss(predicted, y)
         gradient = loss.gradient(predicted, y)
         net.backward(gradient)
         optimizer.step(net)
      accuracy = fizzbuzz_accuracy(101, 1024, net)
      t.set_description(f"fb loss: {epoch_loss:.3f} acc: {accuracy:.2f}")
# Comprueba de nuevo resultados en el conjunto de prueba
print("test results", fizzbuzz_accuracy(1, 101, net))
```

Dropout

Como la mayoría de los modelos de *machine learning*, las redes neuronales tienen tendencia a sobreajustar a sus datos de entrenamiento. Ya hemos visto antes formas de aliviar esto; por ejemplo, en la sección "Regularización" del capítulo 15 penalizábamos los pesos grandes, lo que ayudaba a evitar el sobreajuste.

Una forma común de regularizar redes neuronales es utilizando la técnica del *dropout* (también llamada dilución o abandono). En el momento del entrenamiento, desactivamos aleatoriamente cada neurona (es decir, reemplazamos su salida por 0) con una cierta probabilidad fija. Esto significa que la red no puede aprender a depender de ninguna neurona individual, cosa que parece ayudar con el sobreajuste.

En el momento de la evaluación, no queremos eliminar ninguna neurona, de modo que una capa `Dropout` tendrá que saber si está entrenando o no. Además, en el momento del entrenamiento, la capa `Dropout` solo pasa una

fracción aleatoria de su entrada. Para que su salida pueda compararse durante la evaluación, reduciremos las salidas (de manera uniforme) utilizando esa misma fracción:

```
class Dropout(Layer):
   def __init__(self, p: float) -> None:
      self.p = p
      self.train = True
   def forward(self, input: Tensor) -> Tensor:
      if self.train:
         # Crea una máscara de ceros y unos con la forma de
         # la entrada usando la probabilidad especificada.
         self.mask = tensor_apply(
            lambda _: 0 if random.random() < self.p else 1,
            input)
         # Multiplica por la máscara para eliminar entradas.
         return tensor_combine(operator.mul, input, self.mask)
      else:
         # Durante la evaluación reduce las salidas uniformemente.
         return tensor_apply(lambda x: x * (1–self.p), input)
   def backward(self, gradient: Tensor) -> Tensor:
      if self.train:
         # Solo propaga los gradientes donde mask == 1.
         return tensor_combine(operator.mul, gradient, self.mask)
      else:
         raise RuntimeError("don’t call backward when not in train mode")
```

Emplearemos esto para evitar el sobreajuste en nuestros modelos de *deep learning*.

Ejemplo: MNIST

MNIST[1] es un conjunto de datos de dígitos manuscritos que todo el mundo utiliza para estudiar *deep learning*. Está disponible en un formato binario algo complicado, de modo que instalaremos la librería `mnist` para trabajar con él (sí, en esta parte no empezamos técnicamente “desde cero”).

```
python -m pip install mnist
```

Después, podemos cargar los datos:

```
import mnist
# Esto descargará los datos; cambie esto a donde los quiera.
# (Sí, es una función con argumento 0, eso es lo que la librería espera).
# (Sí, estoy asignando una lambda a una variable, como dije que nunca haría).
mnist.temporary_dir = lambda: '/tmp'
# Cada una de estas funciones descarga primero los datos y devuelve un array
numpy.
# Llamamos a .tolist() porque nuestros "tensores" son solo listas.
train_images = mnist.train_images().tolist()
train_labels = mnist.train_labels().tolist()
assert shape(train_images) == [60000, 28, 28]
assert shape(train_labels) == [60000]
```

Mostremos en pantalla las primeras 100 imágenes de entrenamiento para ver qué aspecto tienen (figura 19.1):

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(10, 10)
for i in range(10):
   for j in range(10):
      # Muestra cada imagen en blanco y negro y oculta los ejes.
      ax[i][j].imshow(train_images[10 * i + j], cmap='Greys')
      ax[i][j].xaxis.set_visible(False)
      ax[i][j].yaxis.set_visible(False)
plt.show()
```

Se puede comprobar que de hecho son dígitos manuscritos.

Nota: Mi primer intento de mostrar las imágenes dio como resultado números amarillos sobre fondo negro. No soy ni lo bastante hábil ni ingenioso para saber que tenía que añadir `cmap=Greys` para conseguir imágenes en blanco y negro; lo busqué en Google y encontré la solución en Stack Overflow. Como científico de datos, seguro que frecuentará mucho este sitio.

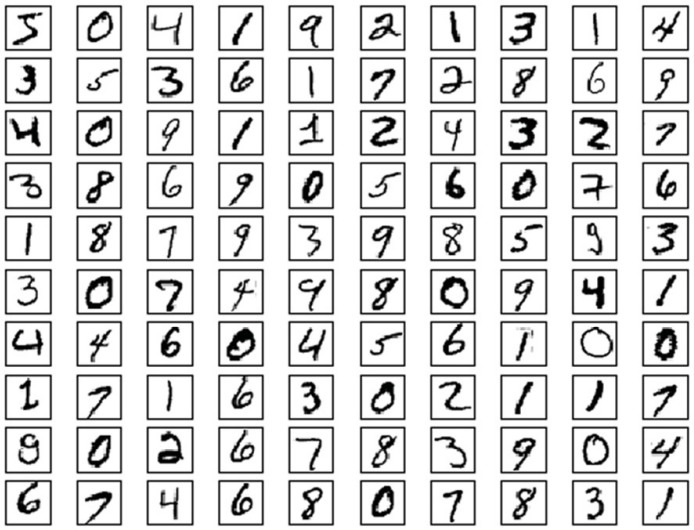


Figura 19.1. Imágenes MNIST.

También tenemos que cargar las imágenes de prueba:

```
test_images = mnist.test_images().tolist()
test_labels = mnist.test_labels().tolist()
assert shape(test_images) == [10000, 28, 28]
assert shape(test_labels) == [10000]
```

Cada imagen tiene 28 x 28 píxeles, pero nuestras capas lineales solo pueden admitir entradas unidimensionales, de modo que simplemente los aplanamos (y dividimos también por 256 para colocarlos entre 0 y 1). Además, nuestra red neuronal entrenará mejor si nuestras entradas son 0 de media, de modo que restamos el valor promedio:

```
# Calcula el valor de píxel promedio
avg = tensor_sum(train_images) / 60000 / 28 / 28
# Vuelve a centrar, redimensiona y aplana
train_images = [[(pixel-avg) / 256 for row in image for pixel in row]
 for image in train_images]
test_images = [[(pixel-avg) / 256 for row in image for pixel in row]
 for image in test_images]
assert shape(train_images) == [60000, 784], "images should be flattened"
assert shape(test_images) == [10000, 784], "images should be flattened"
# Tras centrar, el píxel promedio debería estar muy cerca de 0
assert -0.0001 < tensor_sum(train_images) < 0.0001
```

También queremos codificar con One-Hot-Encode los objetivos, ya que tenemos 10 salidas. Primero, escribimos una función `one_hot_encode`:

```
def one_hot_encode(i: int, num_labels: int = 10) -> List[float]:
return [1.0 if j == i else 0.0 for j in range(num_labels)]
assert one_hot_encode(3) == [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
assert one_hot_encode(2, num_labels=5) == [0, 0, 1, 0, 0]
```

Después, la aplicamos a nuestros datos:

```
train_labels = [one_hot_encode(label) for label in train_labels]
test_labels = [one_hot_encode(label) for label in test_labels]

assert shape(train_labels) == [60000, 10]
assert shape(test_labels) == [10000, 10]
```

Uno de los puntos fuertes de nuestras abstracciones es que podemos utilizar el mismo bucle de entrenamiento/evaluación con distintos modelos. Por tanto, escribamos eso primero. Le pasaremos nuestro modelo, los datos, una función de pérdida y (si estamos entrenando) un optimizador. Hará una pasada por nuestros datos, controlará el rendimiento y (si le hemos pasado un optimizador) actualizará nuestros parámetros:

```
import tqdm
def loop(model: Layer,
     images: List[Tensor],
     labels: List[Tensor],
     loss: Loss,
     optimizer: Optimizer = None) -> None:
   correct = 0                                # Controla número de predicciones
                                              correctas.
   total_loss = 0.0                           # Controla pérdida total.
   with tqdm.trange(len(images)) as t:
      for i in t:
         predicted =                          # Predice.
         model.forward(images[i])
         if argmax(predicted) ==              # Comprueba si
         argmax(labels[i]):
            correct += 1                      # es correcto o no.
         total_loss += loss.loss(predicted,   # Calcula pérdida.
         labels[i])
         # Si estamos entrenando, propaga hacia atrás gradiente y actualiza
         pesos.
         if optimizer is not None:
```

```
            gradient = loss.gradient(predicted, labels[i])
            model.backward(gradient)
            optimizer.step(model)
        # Y actualiza nuestra métrica en la barra de progreso.
        avg_loss = total_loss / (i + 1)
        acc = correct / (i + 1)
        t.set_description(f"mnist loss: {avg_loss:.3f} acc: {acc:.3f}")
```

Como referencia, podemos utilizar nuestra librería de *deep learning* para entrenar un modelo de regresión logístico (multiclase), que no es más que una sola capa lineal seguida de una softmax. Lo que hace este modelo (esencialmente) es buscar 10 funciones lineales tales que, si la entrada representa, digamos, un 5, entonces la 5ª función lineal produce el resultado más grande.

Una pasada por nuestros 60.000 ejemplos de entrenamiento debería bastar para aprender el modelo:

```
random.seed(0)

# La regresión logística es una capa lineal seguida de softmax
model = Linear(784, 10)
loss = SoftmaxCrossEntropy()

# Este optimizador parece funcionar
optimizer = Momentum(learning_rate=0.01, momentum=0.99)

# Entrena con los datos de entrenamiento
loop(model, train_images, train_labels, loss, optimizer)

# Prueba con los datos de prueba (sin optimizador significa solo evaluar)
loop(model, test_images, test_labels, loss)
```

Este código alcanza un 89 % de precisión. Veamos si podemos hacerlo mejor con una red neuronal profunda. Utilizaremos dos capas ocultas, la primera con 30 neuronas y la segunda con 10. Además, emplearemos nuestra activación `Tanh`:

```
random.seed(0)
# Los nombra para poder poner y quitar entrenamiento
dropout1 = Dropout(0.1)
dropout2 = Dropout(0.1)
model = Sequential([
   Linear(784, 30),      # Capa oculta 1: tamaño 30
   dropout1,
```

```
    Tanh(),
    Linear(30, 10),       # Capa oculta 2: tamaño 10
    dropout2,
    Tanh(),
    Linear(10, 10)        # Capa de salida: tamaño 10
])
```

¡Encima podemos utilizar el mismo bucle de entrenamiento!

```
optimizer = Momentum(learning_rate=0.01, momentum=0.99)
loss = SoftmaxCrossEntropy()

# Activa dropout y entrena (¡tarda > 20 minutos en mi portátil!)
dropout1.train = dropout2.train = True
loop(model, train_images, train_labels, loss, optimizer)

# Desactiva dropout y evalúa
dropout1.train = dropout2.train = False
loop(model, test_images, test_labels, loss)
```

Nuestro modelo profundo alcanza una precisión más alta del 92 % en el conjunto de prueba, lo que supone una buena mejora con respecto al modelo logístico sencillo.

El sitio web de MNIST describe distintos modelos que superan a estos. Muchos de ellos podrían implementarse utilizando los mecanismos que hemos desarrollado hasta ahora, pero tardaría demasiado tiempo en entrenar en nuestra estructura de listas como tensores. Algunos de los mejores modelos implican capas convolucionales, que son importantes, pero lamentablemente quedan fuera del alcance de un libro de introducción en ciencia de datos como este.

Guardar y cargar modelos

Se tarda mucho en entrenar estos modelos, de modo que estaría bien si pudiéramos guardarlos de forma que no tengamos que adiestrarlos cada vez. Por suerte, podemos utilizar el módulo `json` para serializar fácilmente los pesos del modelo en un archivo.

Para guardarlos podemos emplear `Layer.params` para recopilar los pesos, meterlos en una lista y usar `json.dump` para guardar esa lista en un archivo:

```
import json
def save_weights(model: Layer, filename: str) -> None:
   weights = list(model.params())
   with open(filename, 'w') as f:
      json.dump(weights, f)
```

Cargar los pesos de nuevo es solo un poco más de trabajo. Utilizamos `json.load` para recuperar la lista de pesos del archivo y asignación de segmentos para fijar los pesos de nuestro modelo. (En particular, esto significa que tenemos que instanciar el modelo nosotros mismos y después cargar los pesos. Un método alternativo sería guardar una representación de la arquitectura del modelo y utilizarla para instanciar el modelo. No es una mala idea, pero requeriría mucho más código y cambios en todas nuestras `Layer`, de modo que nos quedaremos con el método más sencillo).

Antes de cargar los pesos, nos gustaría comprobar que tienen las mismas formas que los parámetros del modelo en los que los estamos cargando (esto es una protección contra, por ejemplo, intentar cargar los pesos para una red profunda guardada en una red más superficial, o cosas parecidas).

```
def load_weights(model: Layer, filename: str) -> None:
   with open(filename) as f:
      weights = json.load(f)
   # Comprueba consistencia
   assert all(shape(param) == shape(weight)
           for param, weight in zip(model.params(), weights))
   # Luego carga usando asignación de slices
   for param, weight in zip(model.params(), weights):
      param[:] = weight
```

Nota: JSON almacena los datos como texto, lo que le convierte en una representación extremadamente ineficaz. En aplicaciones reales probablemente se utilizaría la librería de serialización `pickle`, que serializa cosas en un formato binario más eficaz. En nuestro caso, decidí mantenerlo sencillo y legible.

Se pueden descargar los pesos para las distintas redes que entrenamos de la página web del libro.

Para saber más

El *deep learning* es ahora mismo el tema de moda, y en este capítulo apenas hemos arañado su superficie. Hay muchos libros buenos y entradas de blogs de calidad (y también muchos posts no tan buenos) sobre casi cualquier aspecto del *deep learning* del que quiera aprender.

- El libro de texto preceptivo *Deep Learning*, en `www.deeplearningbook.org`, de Ian Goodfellow, Yoshua Bengio y Aaron Courville (MIT Press), está disponible gratuitamente en la red. Es muy bueno, pero conlleva unas cuantas matemáticas.
- *Deep Learning with Python* de François Chollet, en `www.manning.com/books/deep-learning-with-python` (Manning), es una estupenda introducción a la librería Keras; con él nuestra biblioteca de *deep learning* está muy bien servida.
- Yo mismo utilizo habitualmente PyTorch, en `pytorch.org`, para el *deep learning*. Su sitio web tiene abundante documentación y muchos tutoriales.

---

[1] `http://yann.lecun.com/exdb/mnist/`.

20 Agrupamiento (clustering)

Donde teníamos tales racimos
Que nos volvían salvajes con nobleza, no locos
—Robert Herrick

La mayoría de los algoritmos de este libro son lo que se conoce como algoritmos de aprendizaje supervisado, en el sentido de que empiezan con un conjunto de datos etiquetados, que usan como base para hacer predicciones sobre datos nuevos y sin etiquetar. Pero el agrupamiento, o *clustering*, es un ejemplo de aprendizaje no supervisado, en el que trabajamos con datos enteramente sin etiquetar (o en el que nuestros datos tienen etiquetas, pero las ignoramos).

La idea

Siempre que observamos una fuente de datos, es probable que los datos formen de algún modo grupos o clústeres. Un conjunto de datos que muestre donde viven los millonarios probablemente tenga grupos en lugares como Beverly Hills o Manhattan. Otro que muestre cuántas horas trabaja la gente cada semana tendrá probablemente un agrupamiento de 40 (o también uno de 20 para gente con jornada reducida). Un conjunto de datos demográficos de los votantes registrados forma casi seguro distintas agrupaciones (por ejemplo, "madres del cole", "jubilados aburridos" o "*millennials* desempleados") relevantes para encuestadores y asesores políticos.

A diferencia de algunos de los problemas que hemos visto, en general no existe un agrupamiento "correcto". Un esquema de agrupación alternativo podría reunir parte de los "*millennials* desempleados" con "estudiantes de posgrado" y otra parte con "habitantes de la buhardilla de los padres". Ningún esquema tiene por qué ser más correcto; en realidad, es probable que

cada uno sea más óptimo con respecto a su propia métrica de corrección de sus grupos.

Además, los grupos no se etiquetan solos. Es necesario hacerlo mirando los datos subyacentes de cada uno.

El modelo

Para nosotros, cada `input` será un vector en el espacio de *d* dimensiones, que, como siempre, representaremos como una lista de números. Nuestro objetivo será identificar los grupos de entradas similares y (a veces) encontrar un valor representativo para cada grupo.

Por ejemplo, cada entrada podría ser un vector numérico que represente el título de una entrada de un post, en cuyo caso el objetivo podría ser encontrar grupos de posts similares, quizá con el fin de entender de qué temas van los blogs de nuestros usuarios. Imaginemos si no que tenemos una imagen que contiene miles de colores (`red, green, blue`) y que necesitamos imprimir en pantalla una versión de dicha imagen en 10 colores. El agrupamiento puede ayudarnos a elegir 10 colores que minimicen el "error de color" total.

Uno de los métodos de agrupamiento más sencillos es *k-means*, en el que primero se elige el número de *k* grupos y después el objetivo es hacer particiones de las entradas en conjuntos S_1, ..., S_k, de una forma que minimice la suma total de distancias cuadradas desde cada punto a la media de su grupo asignado.

Hay muchas maneras de asignar *n* puntos a *k* clústeres, lo que significa que hallar un agrupamiento óptimo es un problema realmente complicado. Nos quedaremos con un algoritmo iterativo que suele encontrar un buen agrupamiento:

1. Empieza con un conjunto de *k-means*, que son puntos en el espacio de *d* dimensiones.
2. Asigna cada punto a la media a la que más se acerca.
3. Si no ha cambiado la asignación de ningún punto, se detiene y mantiene los grupos.

4. Si ha cambiado la asignación de algún punto, recalcula las medias y vuelve al paso 2.

Utilizando la función `vector_mean` del capítulo 4, es bastante fácil crear una clase que haga esto.

Para empezar, crearemos una función auxiliar que mida en cuántas coordenadas difieren dos vectores. La utilizaremos para hacer el seguimiento del progreso de nuestro entrenamiento:

```
from scratch.linear_algebra import Vector
def num_differences(v1: Vector, v2: Vector) -> int:
   assert len(v1) == len(v2)
   return len([x1 for x1, x2 in zip(v1, v2) if x1 != x2])
assert num_differences([1, 2, 3], [2, 1, 3]) == 2
assert num_differences([1, 2], [1, 2]) == 0
```

También necesitamos una función que, dados unos vectores y sus asignaciones a clústeres, calcule las medias de los grupos. Puede darse el caso de que alguno de los grupos no tenga puntos asignados. No podemos tomar la media de una colección vacía, de modo que en ese caso elegiremos aleatoriamente uno de los puntos que sirva como “media” de ese grupo:

```
from typing import List
from scratch.linear_algebra import vector_mean
def cluster_means(k: int,
             inputs: List[Vector],
             assignments: List[int]) -> List[Vector]:
   # clusters[i] contiene las entradas cuya asignación es i
   clusters = [[] for i in range(k)]
   for input, assignment in zip(inputs, assignments):
      clusters[assignment].append(input)
   # si un grupo está vacío, usa un punto aleatorio
   return [vector_mean(cluster) if cluster else random.choice(inputs)
        for cluster in clusters]
```

Y ahora estamos listos para codificar nuestro agrupador. Como siempre, usaremos `tqdm` para controlar el progreso, pero en este caso no sabemos cuántas iteraciones harán falta, así que utilizamos `itertools.count`, que crea un iterable infinito, y lo devolveremos con `return` cuando hayamos

terminado:

```
import itertools
import random
import tqdm
from scratch.linear_algebra import squared_distance
class KMeans:
   def __init__(self, k: int) -> None:
      self.k = k                  # número de clústeres
      self.means = None
   def classify(self, input: Vector) -> int:
      """return the index of the cluster closest to the input"""
      return min(range(self.k),
              key=lambda i: squared_distance(input, self.means[i]))
   def train(self, inputs: List[Vector]) -> None:
      # Inicia con asignaciones aleatorias
      assignments = [random.randrange(self.k) for _ in inputs]
      with tqdm.tqdm(itertools.count()) as t:
         for _ in t:
            # Calcula la media y halla nuevas asignaciones
            self.means = cluster_means(self.k, inputs, assignments)
            new_assignments = [self.classify(input) for input in inputs]
            # Comprueba cuántas asignaciones cambiaron y si hemos terminado
            num_changed = num_differences(assignments, new_assignments)
            if num_changed == 0:
               return
            # Si no mantiene las nuevas asignaciones y calcula nuevas medias
            assignments = new_assignments
            self.means = cluster_means(self.k, inputs, assignments)
            t.set_description(f"changed: {num_changed} / {len(inputs)}")
```

Veamos cómo funciona esto.

Ejemplo: Encuentros

Para celebrar el crecimiento de DataSciencester, la vicepresidenta de Recompensas al usuario quiere organizar varios encuentros en persona para los usuarios de su localidad, incluyendo cerveza, pizza y camisetas de DataSciencester. Usted conoce las ubicaciones de todos sus usuarios cercanos (figura 20.1), y ella quiere que elija lugares para los encuentros que permitan

a todos asistir cómodamente.

Dependiendo de cómo se mire, probablemente se ven dos o tres grupos (es fácil hacerlo visualmente porque los datos son bidimensionales. Con más dimensiones sería mucho más complicado hacerlo a ojo).

Imaginemos primero que la vicepresidenta tiene presupuesto suficiente para tres encuentros. Así que va a su ordenador y prueba lo siguiente:

```
random.seed(12)                   # así obtiene el mismo resultado que yo
clusterer = KMeans(k=3)
clusterer.train(inputs)
means = sorted(clusterer.means)     # ordena para la prueba de unidad
assert len(means) == 3
# Revisa que las medias estén cerca de lo que esperamos
assert squared_distance(means[0], [-44, 5]) < 1
assert squared_distance(means[1], [-16, -10]) < 1
assert squared_distance(means[2], [18, 20]) < 1
```

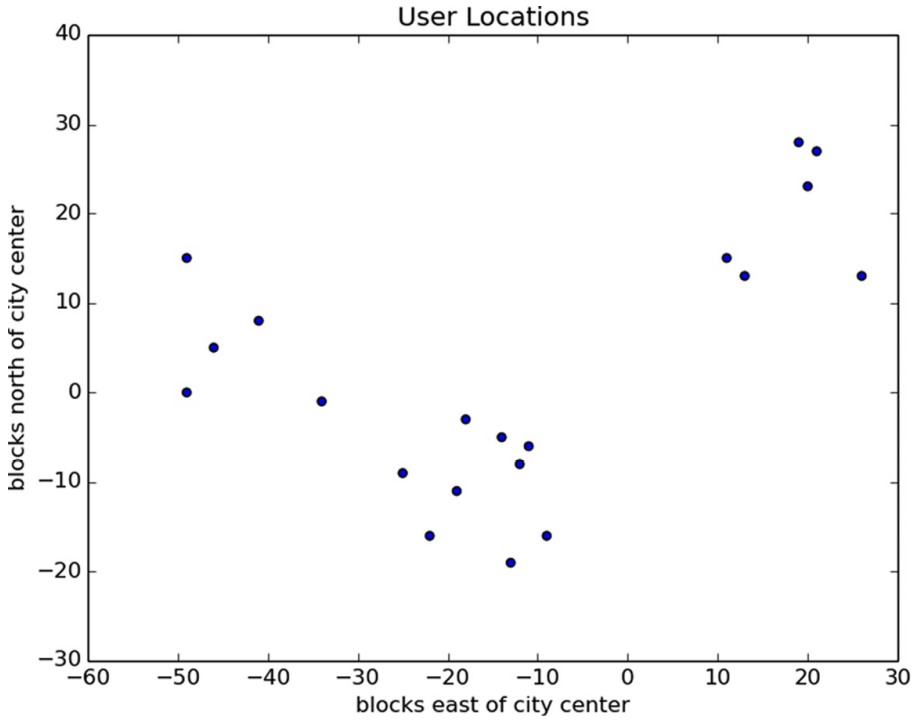


Figura 20.1. Las ubicaciones de los usuarios de su localidad.

Con esto encuentra tres grupos centrados en [-44, 5], [-16, -10] y [18, 20] y busca entonces lugares para los encuentros cercanos a esas ubicaciones (figura 20.2).

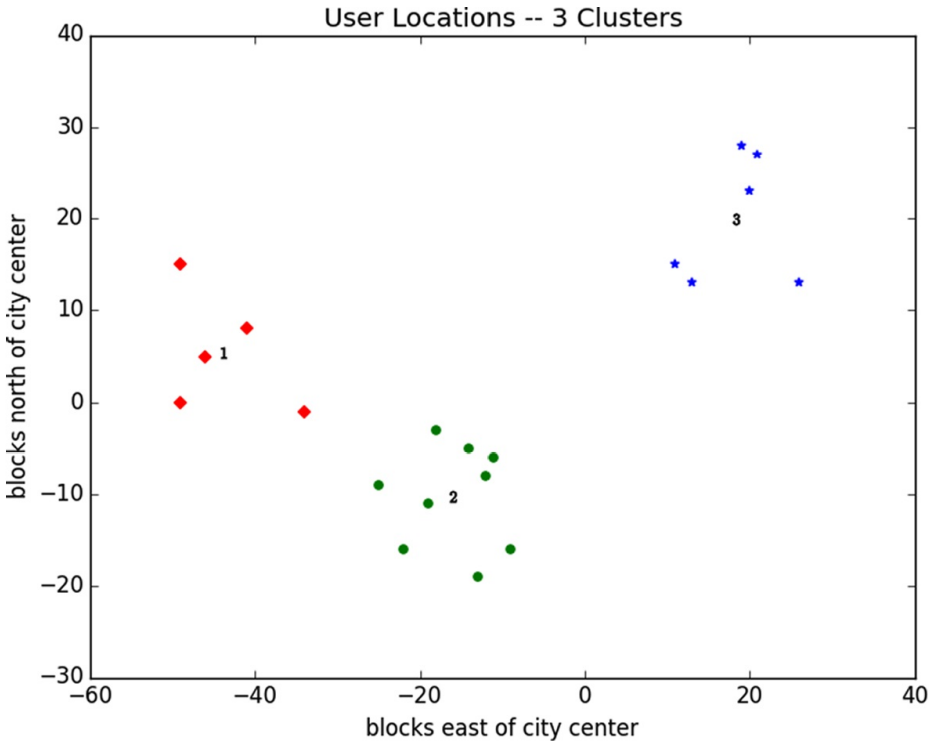


Figura 20.2. Las ubicaciones de los usuarios agrupadas en tres clústeres.

Muestra los resultados a su vicepresidenta, que le informa de que ahora solo tiene presupuesto para dos encuentros.

"Sin problemas", le dice usted:

```
random.seed(0)
clusterer = KMeans(k=2)
clusterer.train(inputs)
means = sorted(clusterer.means)

assert len(means) == 2
assert squared_distance(means[0], [-26, -5]) < 1
assert squared_distance(means[1], [18, 20]) < 1
```

Como se muestra en la figura 20.3, un encuentro aún podría producirse cerca de [18, 20], pero ahora el otro debería tener lugar cerca de [-26, -5].

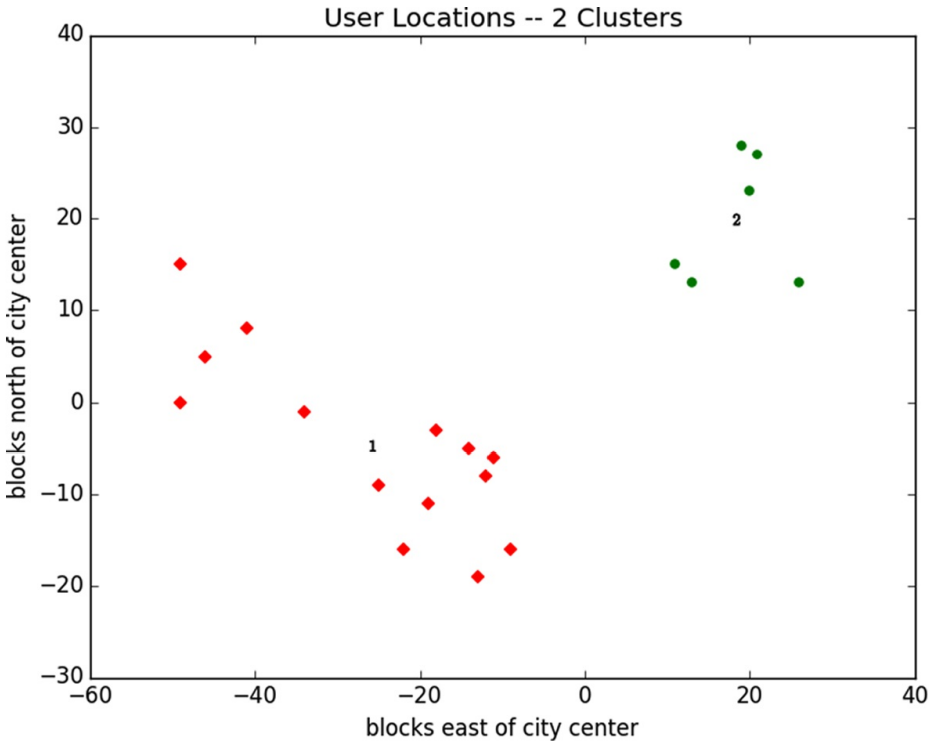


Figura 20.3. Las ubicaciones de los usuarios agrupados en dos clústeres.

Eligiendo k

En el ejemplo anterior, la elección de *k* vino impuesta por factores que estaban fuera de nuestro control. En general, esto no será así. Hay varias formas de elegir un *k*. Una que es razonablemente fácil de entender implica mostrar en pantalla la suma de errores cuadrados (entre cada punto y la media de su grupo) como una función de *k* y observar dónde se "curva" el gráfico:

```
from matplotlib import pyplot as plt
def squared_clustering_errors(inputs: List[Vector], k: int) -> float:
   """finds the total squared error from k-means clustering the inputs"""
   clusterer = KMeans(k)
   clusterer.train(inputs)
   means = clusterer.means
   assignments = [clusterer.classify(input) for input in inputs]
   return sum(squared_distance(input, means[cluster])
           for input, cluster in zip(inputs, assignments))
```

Que podemos aplicar a nuestro ejemplo anterior:

```
# ahora traza desde 1 hasta len(inputs) clústeres

ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]

plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("total squared error")
plt.title("Total Error vs. # of Clusters")
plt.show()
```

Mirando a la figura 20.4, este método está de acuerdo con nuestra medición a ojo original de que tres es el número “correcto” de grupos:

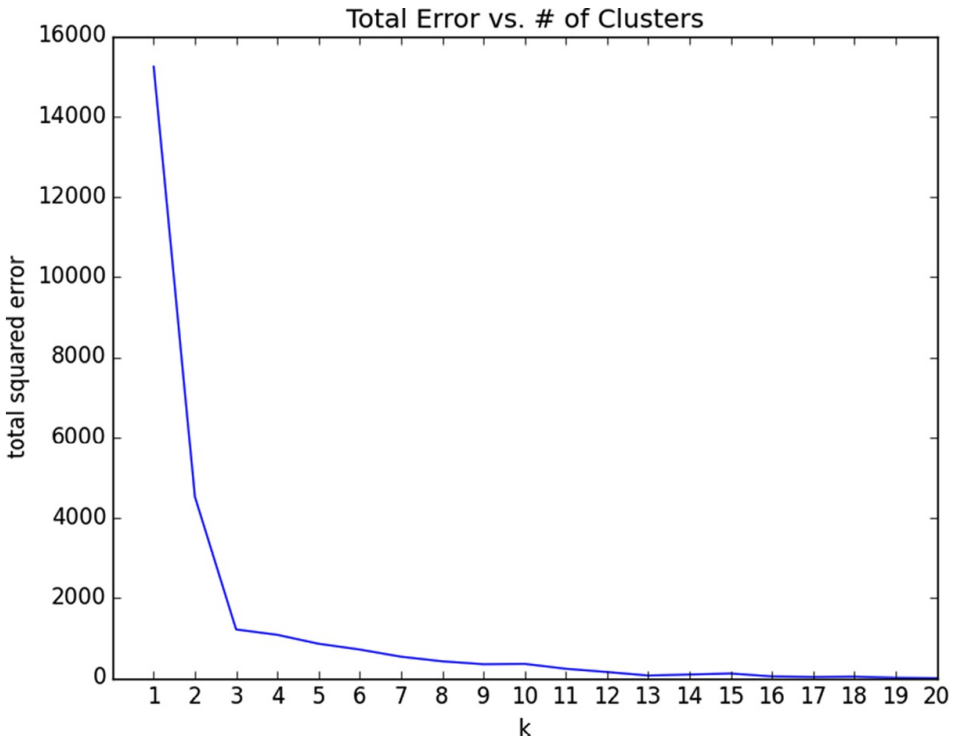


Figura 20.4. Eligiendo un *k*.

Ejemplo: agrupando colores

El vicepresidente de Estilo ha diseñado unas bonitas pegatinas de DataSciencester que le gustaría regalar en los encuentros. Por desgracia, su

impresora de pegatinas puede imprimir como máximo cinco colores por pegatina. Como el vicepresidente de Arte está de año sabático, el de Estilo le pregunta si hay algún modo de que pueda modificar este diseño de modo que contenga solo cinco colores.

Las imágenes de ordenador se pueden representar como *arrays* bidimensionales de píxeles, donde cada píxel es a su vez un vector tridimensional `(red, green, blue)` que indica su color.

Por lo tanto, crear una versión de cinco colores de la imagen implica lo siguiente:

1. Elegir cinco colores.
2. Asignar uno de esos colores a cada píxel.

Resulta que esta es una tarea estupenda para el agrupamiento *k-means*, que puede dividir en particiones los píxeles en cinco grupos en el espacio rojo-verde-azul. Si después coloreamos de nuevo los píxeles de cada grupo con el color promedio, lo tenemos hecho.

Para empezar, necesitamos una forma de cargar una imagen en Python. Podemos hacerlo con matplotlib, si antes instalamos la librería pillow:

```
python -m pip install pillow
```

Después, podemos utilizar `matplotlib.image.imread`:

```
image_path = r"girl_with_book.jpg"       # donde esté su imagen
import matplotlib.image as mpimg
img = mpimg.imread(image_path) / 256     # redimensiona a entre 0 y 1
```

En segundo plano, `img` es un *array* NumPy, pero, para nuestro caso, podemos tratarlo como una lista de listas de listas.

`img[i][j]` es el píxel de la fila *i* y columna *j*, y cada píxel es una lista `[red, green, blue]` de números entre 0 y 1, que indican el color de ese píxel:[1]

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

En particular, podemos obtener una lista combinada de todos los píxeles como:

```
# .tolist() convierte un array NumPy en una lista Python
pixels = [pixel.tolist() for row in img for pixel in row]
```

Y después pasársela a nuestro agrupador:

```
clusterer = KMeans(5)
clusterer.train(pixels)      # puede tardar un poco
```

Una vez haya terminado, tan solo construimos una nueva imagen con el mismo formato:

```
def recolor(pixel: Vector) -> Vector:
   cluster = clusterer.classify(pixel)       # índice del clúster más cercano
   return clusterer.means[cluster]           # media del clúster más cercano
new_img = [[recolor(pixel) for pixel in row] # recolorea esta fila de píxeles
        for row in img]                      # para cada fila de la imagen
```

Y la mostramos, utilizando `plt.imshow`:

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

Es difícil mostrar resultados de color en un libro en blanco y negro, pero en la figura 20.5 pueden verse versiones en escalas de gris de una imagen a todo color y el resultado de utilizar este proceso para reducirla a cinco colores.

Figura 20.5. Imagen original y su decoloración con *5-means*.

Agrupamiento jerárquico de abajo a arriba

Un método alternativo al agrupamiento es "hacer crecer" los grupos de abajo arriba. Podemos hacerlo de la siguiente manera:

1. Convertir cada entrada en su propio grupo de uno.
2. Siempre que queden varios grupos, encontrar los dos más cercanos y combinarlos.

Al final, tendremos un clúster gigante que contendrá todas las entradas. Si controlamos el orden de combinación, podemos recrear cualquier número de grupos descombinándolos. Por ejemplo, si queremos tres clústeres, podemos sencillamente deshacer las dos últimas combinaciones.

Usaremos una representación muy sencilla de los clústeres. Nuestros valores residirán en grupos hoja, que representaremos como `NamedTuples`:

```
from typing import NamedTuple, Union
class Leaf(NamedTuple):
   value: Vector
leaf1 = Leaf([10, 20])
leaf2 = Leaf([30, -15])
```

Los utilizaremos para obtener grupos combinados, que también representaremos como `NamedTuples`:

```
class Merged(NamedTuple):
   children: tuple
   order: int
merged = Merged((leaf1, leaf2), order=1)
Cluster = Union[Leaf, Merged]
```

Nota: Este es otro caso en el que las anotaciones de tipo de Python nos han decepcionado. Querríamos comprobar el tipo de `Merged.children` como `Tuple[Cluster, Cluster]`, pero `mypy` no permite tipos recursivos como este.

Hablaremos en un momento del orden de combinación, pero primero creemos una función auxiliar que devuelva recursivamente todos los valores contenidos en un grupo (posiblemente combinado):

```
def get_values(cluster: Cluster) -> List[Vector]:
   if isinstance(cluster, Leaf):
      return [cluster.value]
   else:
      return [value
           for child in cluster.children
           for value in get_values(child)]
assert get_values(merged) == [[10, 20], [30, -15]]
```

Para combinar los grupos más cercanos, necesitamos una cierta noción de la distancia entre clústeres. Emplearemos la distancia mínima entre elementos de los dos grupos, lo que combina los dos clústeres que están más cerca de tocarse (pero producirá en ocasiones grandes grupos en forma de cadena que no están muy apretados). Si lo que queríamos eran grupos esféricos apretados, podríamos utilizar la distancia máxima, ya que combina los dos grupos que encajan en la bola más pequeña. Ambas son opciones habituales, como lo es la distancia media:

```
from typing import Callable
from scratch.linear_algebra import distance
def cluster_distance(cluster1: Cluster,
                cluster2: Cluster,
                distance_agg: Callable = min) -> float:
   """
   compute all the pairwise distances between cluster1 and cluster2
   and apply the aggregation function _distance_agg_ to the resulting list
   """
   return distance_agg([distance(v1, v2)
                for v1 in get_values(cluster1)
                for v2 in get_values(cluster2)])
```

Nos ayudaremos del orden de combinación para controlar exactamente eso, el orden en el que combinamos. Los números más pequeños representan combinaciones posteriores, lo que significa que, cuando queremos

descombinar clústeres, lo hacemos del orden de combinación más bajo al más alto. Y, como no tienen una propiedad `.order`, crearemos una función auxiliar:

```
def get_merge_order(cluster: Cluster) -> float:
   if isinstance(cluster, Leaf):
      return float('inf')      # nunca se combinó
   else:
      return cluster.order
```

De forma similar, como los grupos `Leaf` no tienen hijos, crearemos otra función asistente para eso:

```
from typing import Tuple
def get_children(cluster: Cluster):
   if isinstance(cluster, Leaf):
      raise TypeError("Leaf has no children")
   else:
      return cluster.children
```

Ya estamos listos para crear el algoritmo de agrupamiento:

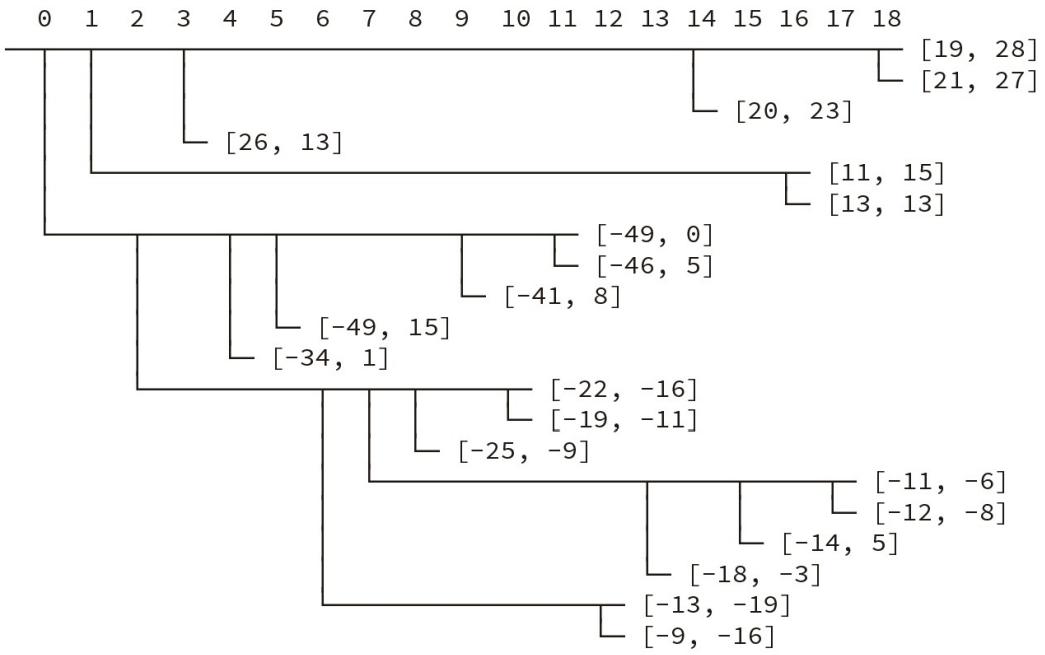
```
def bottom_up_cluster(inputs: List[Vector],
                distance_agg: Callable = min) -> Cluster:
   # Empieza con todas las hojas
   clusters: List[Cluster] = [Leaf(input) for input in inputs]
   def pair_distance(pair: Tuple[Cluster, Cluster]) -> float:
      return cluster_distance(pair[0], pair[1], distance_agg)
   # siempre que quede más de un clúster...
   while len(clusters) > 1:
      # halla los dos grupos más cercanos
      c1, c2 = min(((cluster1, cluster2)
                for i, cluster1 in enumerate(clusters)
                for cluster2 in clusters[:i]),
                key=pair_distance)
      # los quita de la lista de clústeres
      clusters = [c for c in clusters if c != c1 and c != c2]
      # los combina, usando merge_order = nº de clústeres que quedan
      merged_cluster = Merged((c1, c2), order=len(clusters))
      # y añade su combinación
      clusters.append(merged_cluster)
   # cuando solo queda un clúster, lo devuelve
```

```
return clusters[0]
```

Su uso es muy sencillo:

```
base_cluster = bottom_up_cluster(inputs)
```

Lo que produce un agrupamiento que tiene este aspecto:



Los números de arriba indican el “orden de combinación”. Como teníamos 20 entradas, necesitó 19 combinaciones para llegar a este clúster en particular. La primera combinación creó el clúster 18 combinando las hojas [18, 28] y [21, 27], y la última combinación creó el grupo 0.

Si solo quisiéramos dos grupos, dividiríamos en la primera bifurcación (“0”), creando un primer clúster con seis puntos y un segundo con el resto. Para tres grupos, continuaríamos a la segunda bifurcación (“1”), que indica dividir ese primer grupo en el clúster con ([19, 28], [21, 27], [20, 23], [26, 13]) y el clúster con ([11, 15], [13, 13]), y así sucesivamente.

Pero generalmente no vamos a querer dejarnos los ojos con malas representaciones de texto como esta. Lo que haremos será escribir una función que genere cualquier número de grupos realizando el número adecuado de descombinaciones:

```
def generate_clusters(base_cluster: Cluster,
                num_clusters: int) -> List[Cluster]:
  # empieza con una lista con solo el grupo base
  clusters = [base_cluster]
  # siempre que no tengamos aún suficientes grupos...
  while len(clusters) < num_clusters:
    # elige el último combinado de nuestros grupos
    next_cluster = min(clusters, key=get_merge_order)
    # lo quita de la lista
    clusters = [c for c in clusters if c != next_cluster]
    # y añade sus hijos a la lista (es decir, los descombina)
    clusters.extend(get_children(next_cluster))
  # cuando tenemos suficientes clústeres...
  return clusters
```

Así, por ejemplo, si queremos generar tres grupos, podemos hacer esto:

```
three_clusters = [get_values(cluster)
            for cluster in generate_clusters(base_cluster, 3)]
```

Que podemos mostrar fácilmente en pantalla:

```
for i, cluster, marker, color in zip([1, 2, 3],
                         three_clusters,
                         ['D','o','*'],
                         ['r','g','b']):
  xs, ys = zip(*cluster)     # truco mágico de descompresión
  plt.scatter(xs, ys, color=color, marker=marker)
  # pone un número en la media del clúster
  x, y = vector_mean(cluster)
  plt.plot(x, y, marker='$' + str(i) + '$', color='black')
plt.title("User Locations — 3 Bottom-Up Clusters, Min")
plt.xlabel("blocks east of city center")
plt.ylabel("blocks north of city center")
plt.show()
```

Esto da resultados muy distintos a los que daba *k-means*, como muestra la figura 20.6.

Como mencionamos anteriormente, esto es debido a que usar `min` en `cluster_distance` tiende a dar grupos en forma de cadena. Si en su lugar empleamos `max` (que da clústeres apretados), queda igual que el resultado de

3-means (figura 20.7).

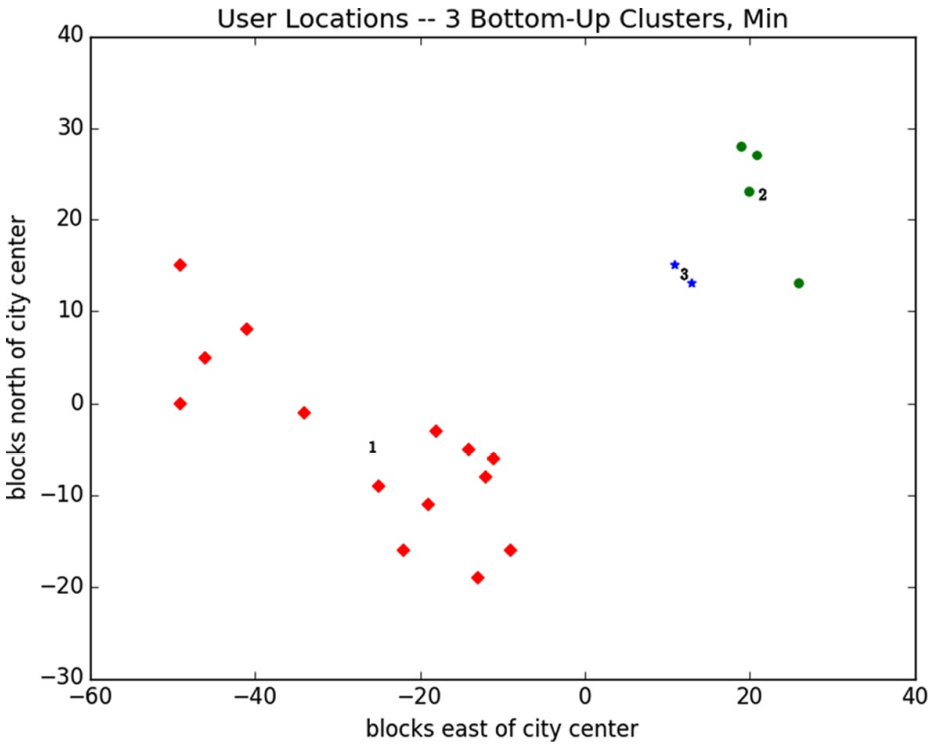


Figura 20.6. Tres grupos de abajo arriba utilizando la distancia mínima.

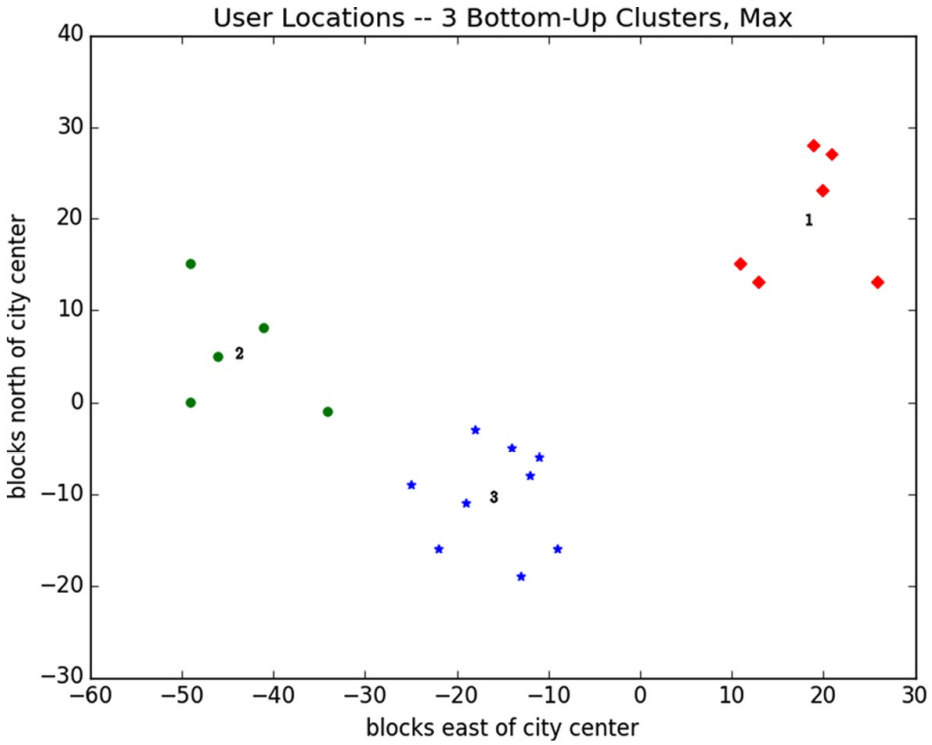


Figura 20.7. Tres grupos de abajo arriba utilizando la distancia máxima.

Nota: La implementación anterior `bottom_up_clustering` es relativamente sencilla, pero también sorprendentemente ineficaz. En particular, recalcula la distancia entre cada par de entradas en cada paso. Una implementación más eficiente podría en su lugar calcular previamente las distancias entre cada par de entradas y después realizar una búsqueda dentro de `cluster_distance`. Y una implementación realmente eficaz también recordaría probablemente las `cluster_distance` del paso anterior.

Para saber más

- scikit-learn tiene un módulo entero, `sklearn.cluster`, en http://scikit-learn.org/stable/modules/clustering.html, que contiene varios algoritmos de agrupamiento incluyendo `Kmeans` y el algoritmo de agrupamiento jerárquico `Ward` (que utiliza un criterio para combinar grupos distinto al nuestro).
- SciPy, en http://www.scipy.org, tiene dos módulos de agrupamiento: `scipy.cluster.vq`, que hace *k-means*, y `scipy.cluster.hierarchy`, que tiene varios algoritmos de agrupamiento jerárquico.

---

[1] https://es.wikipedia.org/wiki/RGB.

21 Procesamiento del lenguaje natural

Han asistido a un gran festín de lenguas, y han robado las sobras.

—William Shakespeare

El procesamiento del lenguaje natural (PLN) se refiere a técnicas de ordenador que involucran el lenguaje. Es un campo extenso, pero veremos algunas técnicas, unas sencillas, pero otras no tanto.

Nubes de palabras

En el capítulo 1 codificamos contadores o recuentos de palabras de los intereses de los usuarios. Un método para visualizar palabras y recuentos es lo que se denomina nubes de palabras, que representan artísticamente las palabras con tamaños proporcionales a sus recuentos.

Pero, en general, los científicos de datos no piensan mucho en nubes de palabras, en gran parte porque la colocación de las palabras no significa nada que no sea “aquí tengo un poco de espacio donde encajar una palabra”.

Si alguna vez se ve obligado a crear una nube de palabras, piense en si puede lograr que los ejes cartesianos transmitan algo. Por ejemplo, imagínese que, por cada palabra de moda relacionada con la ciencia de datos de una cierta colección, tiene dos números entre 0 y 100 (representando el primero la frecuencia con la que aparece en anuncios de empleo y el segundo la frecuencia con la que aparece en currículum vitae):

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
("data science", 60, 70), ("analytics", 90, 3),
("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
("actionable insights", 40, 30), ("think out of the box", 45, 10),
```

```
("self-starter", 30, 50), ("customer focus", 65, 15),
("thought leadership", 35, 35)]
```

El método de la nube de palabras sirve sencillamente para colocar las palabras en una página con una bonita fuente (figura 21.1).

Figura 21.1. Nube de palabras de moda.

Así quedan muy bien, pero en realidad no nos dicen nada. Un enfoque más interesante podría ser distribuirlas de forma que la posición horizontal indique popularidad en los anuncios de empleo y la vertical popularidad en los currículos, lo que produce una visualización que sí transmite algunas ideas (figura 21.2):

```
from matplotlib import pyplot as plt
def text_size(total: int) -> float:
   """equals 8 if total is 0, 28 if total is 200"""
   return 8 + total / 200 * 20
for word, job_popularity, resume_popularity in data:
   plt.text(job_popularity, resume_popularity, word,
 ha='center', va='center',
 size=text_size(job_popularity + resume_popularity))
```

```
plt.xlabel("Popularity on Job Postings")
plt.ylabel("Popularity on Resumes")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

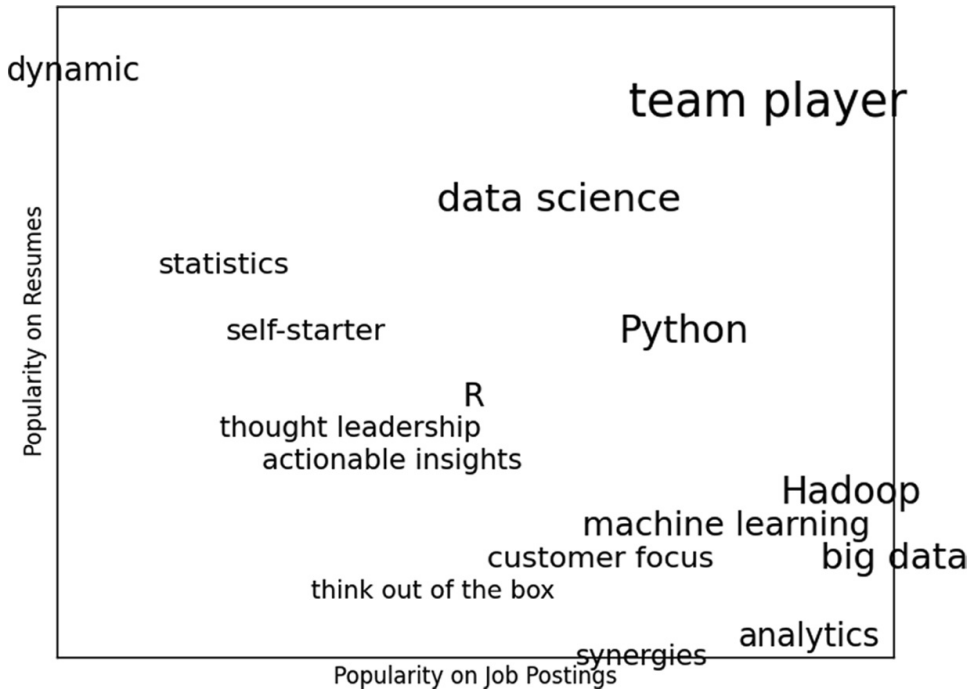


Figura 21.2. Una nube de palabras con más significado (aunque menos bonita).

Modelos de lenguaje n-Gram

La vicepresidenta de Marketing de Motores de Búsqueda de DataSciencester quiere crear miles de páginas web sobre ciencia de datos que hagan que el sitio web aparezca en los puestos más altos de los resultados de búsqueda de términos relacionados con la ciencia de datos (usted intenta explicarle que los algoritmos de motores de búsqueda son tan inteligentes que realmente esto no funcionará, pero se niega a escuchar).

Por supuesto, ella no quiere escribir miles de páginas web ni tampoco pagar a una horda de “estrategas de contenido” que lo hagan. Lo que sí hace es preguntarle a usted si puede generar estas páginas web de algún modo

programando. Para ello, necesitaremos alguna forma de lenguaje de modelado.

Una forma es empezar con una colección de documentos y estudiar un modelo estadístico de lenguaje. En nuestro caso, empezaremos con el ensayo de Mike Loukides *What is data science?*.[1]

Igual que en el capítulo 9, utilizaremos las librerías Requests and Beautiful Soup para recuperar los datos. Hay un par de cuestiones sobre las que vale la pena llamar la atención.

La primera es que los apóstrofos del texto son en realidad el carácter Unicode `u"\u2019"`. Crearemos una función auxiliar para reemplazarlos por comillas normales:

```
def fix_unicode(text: str) -> str:
return text.replace(u"\u2019", "'")
```

La segunda cuestión es que, una vez obtengamos el texto de la página web, nos interesará dividirlo en una secuencia de palabras y puntos (de modo que podamos decir dónde terminan las frases). Podemos hacerlo usando `re.findall`:

```
import re
from bs4 import BeautifulSoup
import requests
url = "https://www.oreilly.com/ideas/what-is-data-science"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
content = soup.find("div", "article-body")# halla la div del article-body
regex = r"[\w']+|[\.]"                    # detecta una palabra o un punto
document = []
for paragraph in content("p"):
   words = re.findall(regex, fix_unicode(paragraph.text))
   document.extend(words)
```

Ciertamente podríamos (y probablemente deberíamos) limpiar más estos datos. Sigue habiendo una cierta cantidad de texto extraño en el documento (por ejemplo, la primera palabra es *Section*), y hemos dividido por puntos a mitad de la frase (por ejemplo, en Web 2.0) y además hay un montón de imágenes y listas por todas partes. Dicho esto, trabajaremos con el

documento tal y como está.

Ahora que tenemos el texto como una secuencia de palabras, podemos modelar el lenguaje de la siguiente manera: dada una palabra inicial (por ejemplo, *book*), miramos todas las palabras que la siguen en el documento de origen. Elegimos aleatoriamente una de ellas para que sea la siguiente, y repetimos el proceso hasta llegar a un punto, que significa el final de la frase. Llamamos a esto modelo de bigrama, ya que está totalmente determinado por las frecuencias de los bigramas (o pares de palabras) de los datos originales.

¿Y qué pasa con la palabra inicial? Podemos elegirla aleatoriamente a partir de las palabras que vayan detrás de un punto. Para empezar, calculemos previamente las posibles transiciones de palabras. Hay que recordar que `zip` se para cuando cualquiera de sus entradas está hecha, de modo que `zip(document, document[1:])` nos da exactamente los pares de elementos consecutivos de `document`:

```
from collections import defaultdict
transitions = defaultdict(list)
for prev, current in zip(document, document[1:]):
   transitions[prev].append(current)
```

Estamos listos para generar frases:

```
def generate_using_bigrams() -> str:
   current = "."                          # significa que la siguiente palabra
                                          iniciará una frase
   result = []
   while True:
      next_word_candidates =                 # bigramas (current, _)
      transitions[current]
      current =                              # elige uno aleatoriamente
      random.choice(next_word_candidates)
      result.append(current)                 # lo añade a los resultados
      if current == ".": return "            # si ".", terminado
      ".join(result)
```

Las frases que produce son un galimatías, pero son el tipo de jerigonza que pondríamos en nuestro sitio web si estuviéramos intentando sonar como científicos de datos. Por ejemplo:

If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.

–Modelo de bigramas

Podemos lograr que las frases no parezcan tanto un guirigay buscando trigramas, es decir, tríos de palabras consecutivas (generalizando más, podríamos buscar n-gramas, formados por *n* palabras consecutivas, pero tres serán bastante para nosotros). Ahora las transiciones dependerán de las dos palabras anteriores:

```
trigram_transitions = defaultdict(list)
starts = []
for prev, current, next in zip(document, document[1:], document[2:]):
   if prev == ".":               # si la "palabra" anterior era un punto
      starts.append(current)     # entonces es palabra inicial
   trigram_transitions[(prev, current)].append(next)
```

Tenga en cuenta que ahora tenemos que controlar las palabras iniciales por separado. Podemos generar frases de una forma bastante parecida:

```
def generate_using_trigrams() -> str:
   current = random.choice(starts)     # elige una palabra inicial aleatoria
   prev = "."                          # y le pone delante un '.'
   result = [current]
   while True:
      next_word_candidates = trigram_transitions[(prev, current)]
      next_word = random.choice(next_word_candidates)
      prev, current = current, next_word
      result.append(current)
      if current == ".":
         return " ".join(result)
```

Este código produce frases mejores como:

In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That's not a question we could even have asked a

few years there has been instrumented.

–Modelo de trigramas

Sin duda, suenan mejor porque en cada paso el proceso de generación tiene menos opciones, y en muchos pasos tan solo una. Esto significa que con frecuencia generamos frases (o al menos oraciones largas) que se vieron textualmente en los datos originales. Tener más datos ayudaría; también funcionaría mejor si recopiláramos n-gramas de distintos ensayos sobre ciencia de datos.

Gramáticas

Un método distinto al lenguaje de modelado es utilizar gramáticas, reglas para generar oraciones aceptables. Es probable que en el colegio aprendiera cuáles son las partes de la oración y cómo se combinan. Por ejemplo, si tuvo un profesor de lengua no demasiado bueno, podría decir que una frase tiene que consistir siempre en un sustantivo seguido de un verbo. Así que, teniendo una lista de sustantivos y verbos, podría generar oraciones de acuerdo con esa regla.

Definiremos una gramática un poco más compleja:

```
from typing import List, Dict
# Escribe un alias para referirse luego a las gramáticas
Grammar = Dict[str, List[str]]
grammar = {
   "_S" : ["_NP _VP"],
   "_NP" : ["_N",
        "_A _NP _P _A _N"],
   "_VP" : ["_V",
        "_V _NP"],
   "_N" : ["data science", "Python", "regression"],
   "_A" : ["big", "linear", "logistic"],
   "_P" : ["about", "near"],
   "_V" : ["learns", "trains", "tests", "is"]
}
```

Me he inventado la norma de que los nombres que empiecen con guion bajo se refieren a reglas que necesitan ser expandidas, y que otros nombres son terminales que no necesitan más procesado.

Así, por ejemplo, `"_S"` es la regla "sentence" (sintagma), que produce una regla `"_NP"` ("noun phrase", es decir, sintagma nominal) seguida de una regla `"_VP"` o "verb phrase" (sintagma verbal).

La regla del sintagma verbal puede producir o bien la regla `"_V"` ("verb", o verbo), o la regla del verbo seguida de la regla del sintagma nominal.

Hay que tener en cuenta que la regla `"_NP"` se contiene a sí misma en una de sus producciones. Las gramáticas pueden ser recursivas, lo que permite hasta a las gramáticas limitadas como esta generar infinitas oraciones diferentes.

¿Cómo generamos frases a partir de esta gramática? Empezaremos con una lista que contiene la regla del sintagma `["_S"]`. Después, ampliaremos repetidamente cada regla sustituyéndola por una de sus producciones elegida aleatoriamente, y pararemos cuando tengamos una lista formada únicamente por terminales.

Por ejemplo, una progresión así podría tener este aspecto:

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

¿Cómo implementamos esto? Pues bien, para empezar, crearemos una sencilla función auxiliar para identificar las terminales:

```
def is_terminal(token: str) -> bool:
   return token[0] != "_"
```

Después tenemos que escribir otra función que convierta una lista de

elementos en una frase. Buscaremos el primer elemento no terminal; si no lo encontramos, significa que tenemos una frase completa y hemos terminado.

Pero, si hallamos un no terminal, entonces elegimos aleatoriamente una de sus producciones. Si dicha producción es un terminal (esto es, una palabra), simplemente reemplazamos el elemento con ella. Si no lo es, entonces es una secuencia de elementos no terminales separados por espacios, que tenemos que dividir con `split` y después unir para formar los verdaderos elementos. En cualquier caso, repetimos el proceso en el nuevo conjunto de elementos obtenido.

Una vez todo montado, tenemos lo siguiente:

```
def expand(grammar: Grammar, tokens: List[str]) -> List[str]:
   for i, token in enumerate(tokens):
      # Si es un elemento terminal, lo salta.
      if is_terminal(token): continue
      # Si no, es un elemento no terminal,
      # y hay que elegir un sustituto aleatoriamente.
      replacement = random.choice(grammar[token])
      if is_terminal(replacement):
         tokens[i] = replacement
      else:
         # El sustituto puede ser, p.ej., "_NP _VP", así
         # hay que dividirlo en espacios y unirlo.
         tokens = tokens[:i] + replacement.split() + tokens[(i+1):]
      # Ahora usa expand en la nueva lista de elementos.
      return expand(grammar, tokens)
   # Si llegamos aquí, teníamos todos los terminales, así que está hecho.
   return tokens
```

Ahora podemos empezar a generar oraciones:

```
def generate_sentence(grammar: Grammar) -> List[str]:
   return expand(grammar, ["_S"])
```

Puede probar a cambiar la gramática (añadiendo más palabras, más reglas o las partes de la oración que desee) hasta estar preparado para generar tantas páginas web como su empresa necesite.

Las gramáticas son mucho más interesantes cuando se utilizan en la otra dirección. Dada una frase, podemos usar una gramática para analizarla, lo que

nos permite identificar sujetos y verbos y nos ayuda a dar sentido a la oración.

Utilizar ciencia de datos para generar texto es un buen truco, pero emplearla para comprender textos es auténtica magia (en el apartado “Para saber más”, al final del capítulo, ofrezco librerías específicas para ello).

Un inciso: muestreo de Gibbs

Generar muestras a partir de ciertas distribuciones es fácil. Podemos conseguir variables aleatorias uniformes con:

```
random.random()
```

Y variables aleatorias normales con:

```
inverse_normal_cdf(random.random())
```

Pero hay algunas distribuciones de las que es más difícil conseguir muestras. El muestreo de Gibbs es una técnica que se emplea para generar muestras a partir de distribuciones multidimensionales cuando solamente conocemos algunas de las distribuciones condicionales.

Por ejemplo, supongamos que tiramos dos dados. Digamos que *x* es el valor que da el primer dado e *y* es la suma de los valores que dan los dos, e imaginemos que queríamos generar muchos pares *(x, y)*. En este caso, es fácil generar directamente las muestras:

```
from typing import Tuple
import random
def roll_a_die() -> int:
   return random.choice([1, 2, 3, 4, 5, 6])
def direct_sample() -> Tuple[int, int]:
   d1 = roll_a_die()
   d2 = roll_a_die()
   return d1, d1 + d2
```

Pero supongamos que solo conocíamos las distribuciones condicionales. La distribución de *y* condicionada por *x* es fácil (si sabemos el valor que da *x*,

y puede ser $x + 1$, $x + 2$, $x + 3$, $x + 4$, $x + 5$ o $x + 6$):

```
def random_y_given_x(x: int) -> int:
   """equally likely to be x + 1, x + 2, ... , x + 6"""
   return x + roll_a_die()
```

La otra dirección es más complicada. Por ejemplo, si sabemos que *y* es 2, entonces *x* tiene que ser obligadamente 1 (ya que la única forma de que al tirar dos dados puedan sumar 2 es que ambos den un 1). Si sabemos que *y* es 3, entonces *x* puede ser tanto 1 como 2. De forma similar, si *y* es 11, entonces *x* tiene que ser 5 o 6:

```
def random_x_given_y(y: int) -> int:
   if y <= 7:
      # si el total es 7 o menos, el primer dado puede dar
      # 1, 2, ..., (total–1)
      return random.randrange(1, y)
   else:
      # si el total es 7 o más, el primer dado puede dar
      # (total–6), (total–5), ..., 6
      return random.randrange(y–6, 7)
```

La forma en la que el muestreo de Gibbs funciona es la siguiente: empezamos con cualesquiera valores (válidos) para *x* e *y*, y después alternamos repetidamente reemplazando *x* con un valor aleatorio elegido en función de *y*, y reemplazando *y* por un valor aleatorio elegido en función de *x*. Tras unas cuantas iteraciones, los valores resultantes de *x* e *y* representarán una muestra de la distribución conjunta no condicional:

```
def gibbs_sample(num_iters: int = 100) -> Tuple[int, int]:
   x, y = 1, 2          # en realidad no importa
   for _ in range(num_iters):
      x = random_x_given_y(y)
      y = random_y_given_x(x)
   return x, y
```

Se puede comprobar que esto da resultados parecidos a la muestra directa:

```
def compare_distributions(num_samples: int = 1000) -> Dict[int, List[int]]:
   counts = defaultdict(lambda: [0, 0])
```

```
for _ in range(num_samples):
   counts[gibbs_sample()][0] += 1
   counts[direct_sample()][1] += 1
return counts
```

Haremos uso de esta técnica en el siguiente apartado.

Modelos de temas

Cuando creamos nuestro sugeridor "Científicos de datos que podría conocer" en el capítulo 1, simplemente buscábamos coincidencias exactas en los intereses declarados por la gente.

Un sistema más sofisticado de comprender los intereses de nuestros usuarios podría tratar de identificar los temas que subyacen bajo esos intereses. La técnica llamada asignación de Dirichlet latente (o LDA, *Latent Dirichlet Allocation*) se emplea habitualmente para identificar temas comunes en una serie de documentos. La aplicaremos a documentos que contienen los intereses de cada usuario.

LDA tiene algunas similitudes con el clasificador Naive Bayes, creado en el capítulo 13, en que asume un modelo probabilístico para los documentos. Pasaremos por alto los detalles matemáticos más complejos, pero, para nuestros fines, el modelo supone que:

- Hay un cierto número fijo *K* de temas.
- Hay una variable aleatoria que asigna a cada tema una distribución de probabilidad asociada a palabras. Pensemos que esta distribución es la probabilidad de ver la palabra *w* dado el tema *k*.
- Existe otra variable aleatoria que asigna a cada documento una distribución de probabilidad según temas. Pensemos que esta distribución es la mezcla de temas del documento *d*.
- Cada palabra de un documento se generó al principio eligiendo aleatoriamente un tema (de la distribución de temas del documento) y después eligiendo aleatoriamente una palabra (de la distribución de palabras del tema).

En particular, tenemos una colección de variables `document`, cada una de las cuales es una `list` de palabras, y tenemos una colección correspondiente de `document_topics` que asigna un `topic` (aquí un número entre 0 y *K* - 1) a cada palabra de cada documento.

Así, la quinta palabra del cuarto documento es:

```
documents[3][4]
```

Y el tema del que esa palabra fue elegida es:

```
document_topics[3][4]
```

Esto define de manera muy explícita la distribución de cada documento según temas, y define de forma implícita la distribución de cada tema según palabras.

Podemos estimar la probabilidad de que el tema 1 produzca una determinada palabra comparando cuántas veces el tema 1 produce esa palabra con cuántas veces el tema 1 produce cualquier palabra (de forma similar, cuando creamos el filtro de *spam* en el capítulo 13, comparamos cuántas veces aparecía cada palabra en los mensajes de *spam* con el número total de palabras que aparecen en los mismos).

Aunque estos temas sean solamente números, podemos darles nombres descriptivos observando las palabras en las que ponen el mayor peso. Solo tenemos que generar de algún modo el `document_topics`. Aquí es donde entra en juego el muestreo de Gibbs.

Empezamos asignando a cada palabra de cada documento un tema totalmente aleatorio. Después, vamos pasando por cada documento una palabra cada vez. Para esa palabra y documento, construimos pesos para cada tema que dependan de la distribución (actual) de temas de ese documento y de la distribución (actual) de palabras para ese tema. A continuación, empleamos esos pesos para extraer un nuevo tema para esa palabra. Si repetimos el proceso muchas veces, terminaremos con una muestra conjunta de la distribución tema-palabra y la distribución documento-tema.

Para empezar, necesitaremos una función que elija aleatoriamente un índice basándose en un conjunto arbitrario de pesos:

```
def sample_from(weights: List[float]) -> int:
```

```
    """returns i with probability weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()      # uniforme entre 0 y total
    for i, w in enumerate(weights):
        rnd -= w                       # devuelve la menor i tal que
        if rnd <= 0: return i          # weights[0] + ... + weights[i] >= rnd
```

Por ejemplo, si le damos pesos `[1, 1, 3]`, entonces una quinta parte del tiempo devolverá 0, una quinta parte del tiempo devolverá 1 y tres quintas partes del tiempo devolverá 2. Escribamos un poco de código de prueba:

```
from collections import Counter
# Tira 1000 veces y cuenta
draws = Counter(sample_from([0.1, 0.1, 0.8]) for _ in range(1000))
assert 10 < draws[0] < 190  # debería ser ~10%, es una prueba muy floja
assert 10 < draws[1] < 190  # debería ser ~10%, es una prueba muy floja
assert 650 < draws[2] < 950 # debería ser ~80%, es una prueba muy floja
assert draws[0] + draws[1] + draws[2] == 1000
```

Nuestros documentos son los intereses de nuestros usuarios, que tienen este aspecto:

```
documents = [
  ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
  ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
  ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
  ["R", "Python", "statistics", "regression", "probability"],
  ["machine learning", "regression", "decision trees", "libsvm"],
  ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
  ["statistics", "probability", "mathematics", "theory"],
  ["machine learning", "scikit-learn", "Mahout", "neural networks"],
  ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
  ["Hadoop", "Java", "MapReduce", "Big Data"],
  ["statistics", "R", "statsmodels"],
  ["C++", "deep learning", "artificial intelligence", "probability"],
  ["pandas", "R", "Python"],
  ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
  ["libsvm", "regression", "support vector machines"]
]
```

E intentaremos encontrar el número de temas:

```
K = 4
```

Para calcular los pesos de muestreo, tendremos que llevar varios recuentos. Creemos primero las estructuras de datos para ellos.

- Cuántas veces se asigna cada tema a cada documento:

```
# una lista de Counter, uno por cada documento
document_topic_counts = [Counter() for _ in documents]
```

- Cuántas veces se asigna cada palabra a cada tema:

```
# una lista de Counter, uno por cada tema
topic_word_counts = [Counter() for _ in range(K)]
```

- El número total de palabras asignado a cada tema:

```
# una lista de números, uno por cada tema
topic_counts = [0 for _ in range(K)]
```

- El número total de palabras contenidas en cada documento:

```
# una lista de números, uno por cada documento
document_lengths = [len(document) for document in documents]
```

- El número de palabras distintas:

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

- Y el número de documentos:

```
D = len(documents)
```

Una vez pobladas estas estructuras, podemos hallar, por ejemplo, el número de palabras de `documents[3]` asociadas al tema 1 de la siguiente manera:

```
document_topic_counts[3][1]
```

Y podemos obtener el número de veces que nlp está asociado al tema 2 del siguiente modo:

```
topic_word_counts[2][“nlp”]
```

Ahora estamos listos para definir nuestras funciones de probabilidad condicional. Como en el capítulo 13, cada una tiene un término suavizante que asegura que cada tema tenga una posibilidad no cero de ser elegido en cualquier documento, y que cada palabra tenga una posibilidad no cero de ser elegida por cualquier tema:

```
def p_topic_given_document(topic: int, d: int, alpha: float = 0.1) -> float:
   """
   The fraction of words in document ‘d’
   that are assigned to ‘topic’ (plus some smoothing)
   """
   return ((document_topic_counts[d][topic] + alpha) /
```

```
        (document_lengths[d] + K * alpha))
def p_word_given_topic(word: str, topic: int, beta: float = 0.1) -> float:
   """
   The fraction of words assigned to 'topic'
   that equal 'word' (plus some smoothing)
   """
   return ((topic_word_counts[topic][word] + beta) /
        (topic_counts[topic] + W * beta))
```

Las utilizaremos para crear los pesos para actualizar temas:

```
def topic_weight(d: int, word: str, k: int) -> float:
   """
   Given a document and a word in that document,
   return the weight for the kth topic
   """
   return p_word_given_topic(word, k) * p_topic_given_document(k, d)
def choose_new_topic(d: int, word: str) -> int:
   return sample_from([topic_weight(d, word, k)
 for k in range(K)])
```

Existen fuertes razones matemáticas por las que `top_weight` está definido así, pero dar más detalles nos llevaría demasiado lejos. Es de esperar que tenga el sentido, al menos intuitivo, de que (dada una palabra y su documento) la probabilidad de cualquier elección de tema dependa tanto de la probabilidad de que ese tema aparezca en el documento como de la probabilidad de que esa palabra pertenezca al tema. Estos son todos los mecanismos que necesitamos. Empezamos asignando cada palabra a un tema aleatorio y poblando nuestros recuentos adecuadamente:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
             for document in documents]
for d in range(D):
   for word, topic in zip(documents[d], document_topics[d]):
      document_topic_counts[d][topic] += 1
      topic_word_counts[topic][word] += 1
      topic_counts[topic] += 1
```

Nuestro objetivo es obtener una muestra conjunta de la distribución temas-palabra y de la distribución documentos-tema. Lo hacemos empleando una

forma de muestreo de Gibbs que utiliza las probabilidades condicionales definidas anteriormente:

```
import tqdm
for iter in tqdm.trange(1000):
   for d in range(D):
      for i, (word, topic) in enumerate(zip(documents[d],
 document_topics[d])):
            # quita esta palabra / tema de los recuentos
            # de modo que no influya en los pesos
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1
            # elige un nuevo tema basado en los pesos
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic
            # y ahora lo añade de nuevo a los recuentos
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

¿Cuáles son los temas? Simplemente son los números 0, 1, 2 y 3. Si queremos que tengan nombre, tendremos que ponérselo nosotros mismos. Veamos las cinco palabras más comunes para cada tema (tabla 21.1):

```
for k, word_counts in enumerate(topic_word_counts):
   for word, count in word_counts.most_common():
      if count > 0:
         print(k, word, count)
```

Tabla 21.1. Palabras más comunes por tema.

Tema 0	Tema 1	Tema 2	Tema 3
Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn

deep learning	probability	Cassandra	machine learning
artificial intelligence	pandas	NoSQL	neural networks

Basándonos en ellas, probablemente yo asignaría estos nombres de tema:

```
topic_names = ["Big Data and programming languages",
           "Python and statistics",
           "databases",
           "machine learning"]
```

Y en este momento podemos ver cómo asigna el modelo temas a los intereses de cada usuario:

```
for document, topic_counts in zip(documents, document_topic_counts):
   print(document)
   for topic, count in topic_counts.most_common():
      if count > 0:
         print(topic_names[topic], count)
      print()
```

Lo que nos da:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

Y así sucesivamente. Visto que necesitamos "and" (conjunciones "y") en algunos de los nombres para nuestros temas, es posible que debamos usar más temas, aunque lo más probable es que no tengamos datos suficientes para estudiarlos con éxito.

Vectores de palabras

Muchos avances recientes en PLN implican *deep learning*. En el resto de este capítulo veremos un par de ellos empleando los mecanismos que

desarrollamos en el capítulo 19.

Una importante innovación tiene que ver con la representación de palabras como vectores de pocas dimensiones. Estos vectores se pueden comparar, sumar, introducir en modelos de *machine learning* o cualquier cosa que se nos ocurra hacer con ellos. Normalmente tienen buenas propiedades; por ejemplo, las palabras parecidas tienden a tener vectores parecidos; es decir, el vector de palabra para *big* es bastante cercano al vector de palabra para *large*, de forma que un modelo que trabaje con vectores de palabras puede (hasta cierto punto) manejar perfectamente los sinónimos.

Los vectores también exhibirán con frecuencia propiedades aritméticas estupendas. Por ejemplo, en algunos modelos, si se toma el vector para *king*, se resta el vector para *man* y se suma el vector para *woman*, se termina con un vector que se parece mucho al vector para *queen*. Puede resultar interesante sopesar lo que esto significa sobre lo que los vectores de palabras realmente "aprenden", aunque ahora mismo no podemos dedicar tiempo a eso.

Conseguir vectores como estos para un gran vocabulario de palabras es una tarea complicada, así que lo normal será que los aprendamos a partir de una colección de textos. Existen un par de esquemas distintos, pero a un nivel alto la tarea suele ser algo parecido a esto:

1. Conseguir un montón de texto.
2. Crear un conjunto de datos en los que el objetivo sea predecir una palabra dadas otras cercanas (o, al contrario, predecir palabras cercanas dada una determinada).
3. Adiestrar a una red neuronal para que maneje bien esta tarea.
4. Utilizar los estados internos de la red neuronal entrenada como los vectores de palabras.

En particular, como la tarea es predecir una palabra dadas otras cercanas, las palabras que aparecen en contextos similares (y por eso tienen palabras cercanas similares) deben tener estados internos parecidos y, por lo tanto, vectores de palabras también parecidos.

Aquí mediremos la "similitud" utilizando la similitud de coseno, que es un

número entre -1 y 1 que mide el grado hacia el que dos vectores apuntan en la misma dirección:

```
from scratch.linear_algebra import dot, Vector
import math
def cosine_similarity(v1: Vector, v2: Vector) -> float:
   return dot(v1, v2) / math.sqrt(dot(v1, v1) * dot(v2, v2))
assert cosine_similarity([1., 1, 1], [2., 2, 2]) == 1, "same direction"
assert cosine_similarity([-1., -1], [2., 2]) == -1, "opposite direction"
assert cosine_similarity([1., 0], [0., 1]) == 0, "orthogonal"
```

Estudiemos algunos vectores de palabras para ver cómo funciona esto.

Para empezar, necesitaremos un conjunto de datos de muestra. Los vectores de palabras habitualmente utilizados suelen derivarse de entrenar con millones o incluso miles de millones de palabras. Como nuestra librería de muestra no puede admitir tantos datos, crearemos para ella un conjunto de datos artificial con una determinada estructura:

```
colors = ["red", "green", "blue", "yellow", "black", ""]
nouns = ["bed", "car", "boat", "cat"]
verbs = ["is", "was", "seems"]
adverbs = ["very", "quite", "extremely", ""]
adjectives = ["slow", "fast", "soft", "hard"]
def make_sentence() -> str:
   return " ".join([
      "The",
      random.choice(colors),
      random.choice(nouns),
      random.choice(verbs),
      random.choice(adverbs),
      random.choice(adjectives),
      "."
   ])
NUM_SENTENCES = 50
random.seed(0)
sentences = [make_sentence() for _ in range(NUM_SENTENCES)]
```

Esto generará muchas frases con estructura parecida, pero palabras distintas, por ejemplo, “*The green boat seems quite slow*”. Dada esta configuración, los colores aparecerán principalmente en contextos “similares”, al igual que los sustantivos, etc. Así, si hacemos un buen trabajo

asignando vectores de palabras, los colores deberían obtener vectores similares, y así sucesivamente.

> **Nota:** En la práctica, probablemente tenga una colección de millones de oraciones, en cuyo caso dispondría de "suficiente" contexto de las frases tal y como son. Aquí, con solo 50 oraciones, tenemos que formarlas de una forma un tanto artificial.

Como ya hemos dicho antes, tenemos que codificar con One-Hot-Encode nuestras palabras, lo que significa que hay que convertirlas en ID. Introduciremos una clase `Vocabulary` para que controle este mapeado:

```
from scratch.deep_learning import Tensor
class Vocabulary:
   def __init__(self, words: List[str] = None) -> None:
      self.w2i: Dict[str, int] = {}      # mapeando word -> word_id
      self.i2w: Dict[int, str] = {}      # mapeando word_id -> word
      for word in (words or []):         # Si hay palabras,
         self.add(word)                  # las añade.
   @property
   def size(self) -> int:
      """how many words are in the vocabulary"""
      return len(self.w2i)
   def add(self, word: str) -> None:
      if word not in self.w2i:           # Si la palabra es nueva:
         word_id = len(self.w2i)         # Halla el siguiente id.
         self.w2i[word] = word_id        # Añade al mapeado word -> word_id.
         self.i2w[word_id] = word        # Añade al mapeado word_id -> word.
   def get_id(self, word: str) -> int:
      """return the id of the word (or None)"""
      return self.w2i.get(word)
   def get_word(self, word_id: int) -> str:
      """return the word with the given id (or None)"""
      return self.i2w.get(word_id)
   def one_hot_encode(self, word: str) -> Tensor:
      word_id = self.get_id(word)
      assert word_id is not None, f"unknown word {word}"
      return [1.0 if i == word_id else 0.0 for i in range(self.size)]
```

Todas estas cosas se podrían hacer manualmente, pero es práctico tenerlas ya en una clase, que probablemente deberíamos probar:

```
vocab = Vocabulary(["a", "b", "c"])
assert vocab.size == 3,                "there are 3 words in the vocab"
assert vocab.get_id("b") == 1,         "b should have word_id 1"
assert vocab.one_hot_encode("b") == [0, 1, 0]
assert vocab.get_id("z") is None,      "z is not in the vocab"
assert vocab.get_word(2) == "c",       "word_id 2 should be c"
vocab.add("z")
assert vocab.size == 4,                "now there are 4 words in the vocab"
assert vocab.get_id("z") == 3,         "now z should have id 3"
assert vocab.one_hot_encode("z") == [0, 0, 0, 1]
```

También tendríamos que escribir sencillas funciones auxiliares para guardar y cargar un vocabulario, igual que lo hacemos para nuestros modelos de *deep learning*:

```
import json
def save_vocab(vocab: Vocabulary, filename: str) -> None:
   with open(filename, 'w') as f:
      json.dump(vocab.w2i, f)      # Solo tiene que guardar w2i
def load_vocab(filename: str) -> Vocabulary:
   vocab = Vocabulary()
   with open(filename) as f:
      # Carga w2i y genera i2w a partir de él
      vocab.w2i = json.load(f)
      vocab.i2w = {id: word for word, id in vocab.w2i.items()}
   return vocab
```

Utilizaremos un modelo de vector de palabra llamado *skip-gram*, que admite como entrada una palabra y genera probabilidades sobre qué palabras es más probable que se vean cerca de esta. Le introduciremos pares de entrenamiento `(word, nearby_word)` e intentaremos minimizar la pérdida `SoftmaxCrossEntropy`.

Nota: Otro modelo típico, CBOW o *continuous-bag-of-words* (bolsa continua de palabras), toma las palabras cercanas como entrada y trata de predecir la palabra original.

Diseñemos nuestra red neuronal. En su núcleo habrá una capa incrustada

que toma como entrada la ID de una palabra y devuelve un vector de palabras. Para esto podemos utilizar una tabla de consulta.

Pasaremos después el vector de palabras a una capa `Linear` con el mismo número de salidas que palabras tenemos en nuestro vocabulario. Igual que antes, utilizaremos `softmax` para convertir estas salidas en probabilidades según palabras cercanas. Cuando empleemos descenso de gradiente para entrenar el modelo, estaremos actualizando los vectores de la tabla de consulta. En cuanto hayamos terminado de entrenarlo, esta tabla nos dará nuestros vectores de palabras.

Creemos la capa incrustada. En la práctica, podríamos querer incrustar elementos que no sean palabras, de modo que construiremos una capa `Embedding` más general (más tarde escribiremos una subclase `TextEmbedding` destinada específicamente a vectores de palabras).

Proporcionaremos a su constructor el número y la dimensión de nuestros vectores incrustados, de forma que pueda crear las incrustaciones de palabras (que inicialmente serán elementos normales aleatorios estándar):

```
from typing import Iterable
from scratch.deep_learning import Layer, Tensor, random_tensor, zeros_like
class Embedding(Layer):
   def __init__(self, num_embeddings: int, embedding_dim: int) -> None:
      self.num_embeddings = num_embeddings
      self.embedding_dim = embedding_dim
      # Un vector de tamaño embedding_dim para cada incrustación deseada
      self.embeddings = random_tensor(num_embeddings, embedding_dim)
      self.grad = zeros_like(self.embeddings)
      # Guarda último id de entrada
      self.last_input_id = None
```

En nuestro caso solo incrustaremos una palabra cada vez. No obstante, en otros modelos nos podría interesar integrar una secuencia de palabras y recuperar una secuencia de vectores de palabras (por ejemplo, si quisiéramos entrenar el modelo CBOW descrito anteriormente). Así, un diseño alternativo tomaría secuencias de identificadores de palabras. Nos quedaremos con una cada vez, para simplificar las cosas.

```
   def forward(self, input_id: int) -> Tensor:
```

```
    """Just select the embedding vector corresponding to the input id"""
    self.input_id = input_id     # recuerde para su uso en retropropagación
    return self.embeddings[input_id]
```

Para el paso atrás obtendremos un gradiente que corresponde al vector incrustado elegido, y tendremos que construir el gradiente correspondiente para `self.embeddings`, que es cero por cada incrustación distinta de la elegida:

```
def backward(self, gradient: Tensor) -> None:
    # Pone a cero el gradiente correspondiente a la última entrada.
    # Esto es mucho más barato que crear un nuevo tensor cero cada vez.
    if self.last_input_id is not None:
        zero_row = [0 for _ in range(self.embedding_dim)]
        self.grad[self.last_input_id] = zero_row
    self.last_input_id = self.input_id
    self.grad[self.input_id] = gradient
```

Como tenemos parámetros y gradientes, necesitamos anular estos métodos:

```
def params(self) -> Iterable[Tensor]:
    return [self.embeddings]
def grads(self) -> Iterable[Tensor]:
    return [self.grad]
```

Como dijimos antes, nos interesa tener una subclase específica para vectores de palabras. En tal caso, nuestro número de incrustaciones está determinado por nuestro vocabulario, de modo que simplemente pasémoslo en su lugar:

```
class TextEmbedding(Embedding):
    def __init__(self, vocab: Vocabulary, embedding_dim: int) -> None:
        # Llama a la superclase constructor
        super().__init__(vocab.size, embedding_dim)
        # Y guarda vocab
        self.vocab = vocab
```

Los otros métodos incorporados funcionarán tal cual, pero añadiremos un par de métodos más específicos para trabajar con texto. Por ejemplo, nos

gustaría poder recuperar el vector para una determinada palabra (esto no es parte de la interfaz `Layer`, pero siempre somos libres de añadir métodos adicionales para capas específicas como queramos).

```
def __getitem__(self, word: str) -> Tensor:
  word_id = self.vocab.get_id(word)
  if word_id is not None:
    return self.embeddings[word_id]
  else:
    return None
```

Este método dunder nos permitirá recuperar vectores de palabras utilizando el indexado:

```
word_vector = embedding[“black”]
```

Y también nos gustaría que la capa incrustada nos diera las palabras más cercanas a una determinada:

```
def closest(self, word: str, n: int = 5) -> List[Tuple[float, str]]:
  """Returns the n closest words based on cosine similarity"""
  vector = self[word]
  # Calcula pares (similarity, other_word), y ordena primero los más
  similares
  scores = [(cosine_similarity(vector, self.embeddings[i]), other_word)
         for other_word, i in self.vocab.w2i.items()]
  scores.sort(reverse=True)
  return scores[:n]
```

Nuestra capa integrada solo proporciona vectores, que podemos introducir en una capa `Linear`.

Ahora estamos listos para montar nuestros datos de entrenamiento. Para cada palabra de entrada, elegiremos como palabras objetivo las dos situadas a su izquierda y las dos a su derecha.

Empecemos poniendo en minúsculas las frases y dividiéndolas en palabras:

```
import re
# No es una gran expresión regular, pero funciona en nuestros datos.
tokenized_sentences = [re.findall("[a-z]+|[.]", sentence.lower())
```

```
 for sentence in sentences]
```

Momento en el cual podemos construir un vocabulario:

```
# Crea un vocabulario (o sea, un mapeado word -> word_id) según nuestro texto.
vocab = Vocabulary(word
             for sentence_words in tokenized_sentences
             for word in sentence_words)
```

Y ahora podemos crear datos de entrenamiento:

```
from scratch.deep_learning import Tensor, one_hot_encode
inputs: List[int] = []
targets: List[Tensor] = []
for sentence in tokenized_sentences:
   for i, word in enumerate(sentence):       # Por cada palabra
      for j in [i–2, i–1, i + 1, i + 2]:     # toma las posiciones cercanas
         if 0 <= j < len(sentence):          # que no están fuera de límites
            nearby_word = sentence[j]        # y obtiene esas palabras.
               # Añade una entrada que es la word_id original
               inputs.append(vocab.get_id(word))
               # Añade un target, la palabra cercana con one-hot-encode
               targets.append(vocab.one_hot_encode(nearby_word))
```

Con las herramientas que hemos construido, ahora es fácil crear nuestro modelo:

```
from scratch.deep_learning import Sequential, Linear
random.seed(0)
EMBEDDING_DIM = 5                # parece un buen tamaño
# Define la capa incrustada por separado, así podemos hacer referencia a ella.
embedding = TextEmbedding(vocab=vocab, embedding_dim=EMBEDDING_DIM)
model = Sequential([
   # Dada una palabra (como un vector de word_id), consulta su incrustación.
   embedding,
   # Y usa una capa lineal para calcular marcadores para "palabras cercanas".
   Linear(input_dim=EMBEDDING_DIM, output_dim=vocab.size)
])
```

Utilizando el código creado en el capítulo 19, es fácil entrenar nuestro modelo:

```
from scratch.deep_learning import SoftmaxCrossEntropy, Momentum,
GradientDescent
loss = SoftmaxCrossEntropy()
optimizer = GradientDescent(learning_rate=0.01)
for epoch in range(100):
   epoch_loss = 0.0
   for input, target in zip(inputs, targets):
      predicted = model.forward(input)
      epoch_loss += loss.loss(predicted, target)
      gradient = loss.gradient(predicted, target)
      model.backward(gradient)
      optimizer.step(model)
   print(epoch, epoch_loss)                 # Imprime la pérdida
   print(embedding.closest("black"))        # y algunas palabras más cercanas
   print(embedding.closest("slow"))         # así podemos ver lo que se está
   print(embedding.closest("car"))          # aprendiendo.
```

Mientras vemos entrenar el modelo, podemos observar cómo se acercan unos a otros tanto los colores como los adjetivos y los sustantivos.

En cuanto el modelo esté entrenado, resulta divertido explorar las palabras más parecidas:

```
pairs = [(cosine_similarity(embedding[w1], embedding[w2]), w1, w2)
 for w1 in vocab.w2i
 for w2 in vocab.w2i
 if w1 < w2]
pairs.sort(reverse=True)
print(pairs[:5])
```

Lo que (para mí) da como resultado:

```
[(0.9980283554864815, 'boat', 'car'),
 (0.9975147744587706, 'bed', 'cat'),
 (0.9953153441218054, 'seems', 'was'),
 (0.9927107440377975, 'extremely', 'quite'),
 (0.9836183658415987, 'bed', 'car')]
```

(Obviamente *bed* y *cat* no son realmente palabras parecidas, pero en nuestras frases de entrenamiento lo parecen, y eso es lo que está capturando el modelo).

También podemos extraer los primeros dos componentes principales y mostrarlos en pantalla:

```
from scratch.working_with_data import pca, transform
import matplotlib.pyplot as plt
# Extrae los dos primeros componentes principales y transforma los vectores de
palabras
components = pca(embedding.embeddings, 2)
transformed = transform(embedding.embeddings, components)
# Dispersa los puntos (y los hace blancos para que sean "invisibles")
fig, ax = plt.subplots()
ax.scatter(*zip(*transformed), marker='.', color='w')
# Añade anotaciones para cada palabra en su posición transformada
for word, idx in vocab.w2i.items():
   ax.annotate(word, transformed[idx])
# Y oculta los ejes
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.show()
```

Lo que demuestra que palabras similares están de hecho agrupadas (figura 21.3):

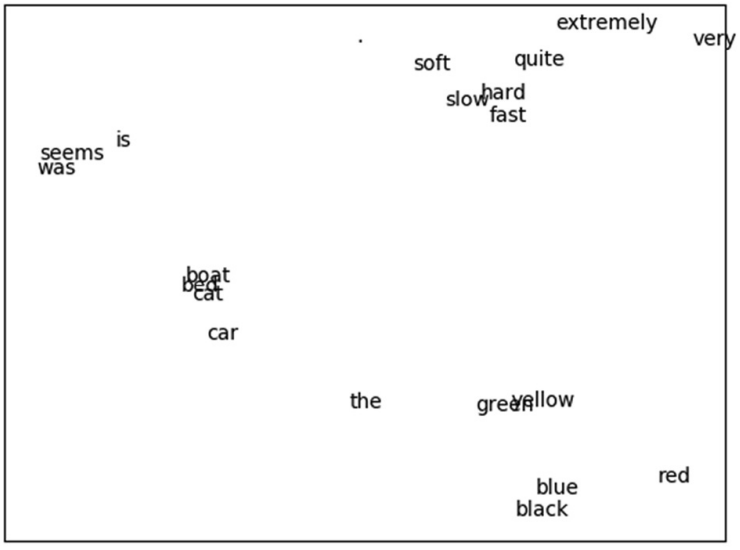


Figura 21.3. Vectores de palabras.

Si le parece interesante, no es difícil entrenar vectores de palabras CBOW.

Pero antes hay que trabajar un poco. Primero, hace falta modificar la capa `Embedding`, de forma que tome como entrada una lista de identificadores y dé como salida una lista de vectores incrustados. Después habrá que crear una nueva capa (`Sum?`) que tome una lista de vectores y devuelva su suma.

Cada palabra representa un ejemplo de entrenamiento, cuya entrada está conformada por los identificadores de las palabras de alrededor, y el objetivo es la aplicación de one-hot-encode a la propia palabra.

La capa `Embedding` modificada convierte las palabras de alrededor en una lista de vectores, la nueva capa `Sum` colapsa la lista de vectores hasta formar un solo vector, y después una capa `Linear` puede producir marcadores a los que se puede aplicar `softmax` para obtener una distribución que represente "la mayoría de las palabras probables, dado este contexto".

El modelo CBOW me pareció más difícil de entrenar que el modelo skip-gram, pero le animo a que lo intente.

Redes neuronales recurrentes

Los vectores de palabras que desarrollamos en el apartado anterior se utilizan con frecuencia como entradas de redes neuronales. Para hacer esto, supone un desafío el hecho de que las frases tengan longitudes diferentes: se podría pensar en una oración de tres palabras como un tensor `[3, embedding_dim]` y una oración de 10 palabras como un tensor `[10, embedding_dim]`. Para poder, por ejemplo, pasarlas a una capa `Linear`, tenemos que hacer algo con respecto a esa primera dimensión de longitud variable.

Una opción es usar una capa `Sum` (o una variante que tome la media); sin embargo, el orden de las palabras de una oración suele ser importante para su significado. Por usar un ejemplo habitual, "*dog bites man*" y "*man bites dog*" son historias muy distintas.

Otra forma de manejar esto es utilizar redes neuronales recurrentes o RNN (*Recurrent Neural Networks*), que tienen un estado oculto que mantienen

entre entradas. En el caso más sencillo, cada entrada se combina con el estado oculto actual para producir un resultado, que se emplea después como el nuevo estado oculto. Ello permite a dichas redes “recordar” (en cierto sentido) las entradas que hemos visto, y llegar a un resultado final que depende de todas las entradas y su orden.

Crearemos la capa RNN más sencilla posible, que aceptará una sola entrada (correspondiente a, por ejemplo, una sola palabra de una frase o a un solo carácter de una palabra), y que mantendrá su estado oculto entre llamadas.

Recordemos que nuestra capa `Linear` tenía algunos pesos, `w`, y un sesgo, `b`. Tomó un `input` de vector y produjo un vector distinto como `output` utilizando la lógica:

```
output[o] = dot(w[o], input) + b[o]
```

Aquí es donde incorporaremos nuestro estado oculto, de modo que tendremos dos series de pesos, una para aplicar a la entrada y otra para aplicar al estado `hidden` anterior:

```
output[o] = dot(w[o], input) + dot(u[o], hidden) + b[o]
```

A continuación, usaremos el vector `output` como nuevo valor de `hidden`. No es un gran cambio, pero permitirá a nuestras redes hacer cosas estupendas.

```
from scratch.deep_learning import tensor_apply, tanh
class SimpleRnn(Layer):
   """Just about the simplest possible recurrent layer."""
   def __init__(self, input_dim: int, hidden_dim: int) -> None:
      self.input_dim = input_dim
      self.hidden_dim = hidden_dim
      self.w = random_tensor(hidden_dim, input_dim, init='xavier')
      self.u = random_tensor(hidden_dim, hidden_dim, init='xavier')
      self.b = random_tensor(hidden_dim)
      self.reset_hidden_state()
   def reset_hidden_state(self) -> None:
      self.hidden = [0 for _ in range(self.hidden_dim)]
```

Podemos ver que empezamos el estado oculto como un vector de ceros, y

proporcionamos una función a la que la gente que utiliza la red puede llamar para reinicializar el estado oculto.

Dada esta configuración, la función `forward` es razonablemente directa (lo es al menos si somos capaces de recordar y comprender cómo funcionaba nuestra capa `Linear`):

```
def forward(self, input: Tensor) -> Tensor:
   self.input = input                          # Guarda la entrada y el estado
                                               oculto
   self.prev_hidden = self.hidden              # anterior para usar en
                                               retropropagación.
   a = [(dot(self.w[h], input) +               # pesos en input
               dot(self.u[h],                  # pesos en hidden
               self.hidden) +
               self.b[h])                      # sesgo
      for h in range(self.hidden_dim)]
   self.hidden = tensor_apply(tanh,   # Aplica activación tanh
   a)
   return self.hidden                 # y devuelve el resultado.
```

El paso `backward` es similar al de nuestra capa `Linear`, excepto que necesita calcular un conjunto de gradientes adicional para los pesos `u`:

```
def backward(self, gradient: Tensor):
   # Retropropaga por tanh
   a_grad = [gradient[h] * (1–self.hidden[h] ** 2)
          for h in range(self.hidden_dim)]
   # b tiene el mismo gradiente que a
   self.b_grad = a_grad
   # Cada w[h][i] se multiplica por input[i] y se suma a a[h],
   # de modo que cada w_grad[h][i] = a_grad[h] * input[i]
   self.w_grad = [[a_grad[h] * self.input[i]
                        for i in range(self.input_dim)]
             for h in range(self.hidden_dim)]
   # Cada u[h][h2] se multiplica por hidden[h2] y se suma a a[h],
   # de modo que cada u_grad[h][h2] = a_grad[h] * prev_hidden[h2]
   self.u_grad = [[a_grad[h] * self.prev_hidden[h2]
                        for h2 in range(self.hidden_dim)]
             for h in range(self.hidden_dim)]
   # Cada input[i] se multiplica por cada w[h][i] y se suma a a[h],
   # de modo que cada input_grad[i] = sum(a_grad[h] * w[h][i] para h en ...)
   return [sum(a_grad[h] * self.w[h][i] for h in range(self.hidden_dim))
        for i in range(self.input_dim)]
```

Y por último necesitamos anular los métodos `param` y `grad`:

```
def params(self) -> Iterable[Tensor]:
   return [self.w, self.u, self.b]
def grads(self) -> Iterable[Tensor]:
   return [self.w_grad, self.u_grad, self.b_grad]
```

Advertencia: Esta “sencilla” RNN lo es tanto, que probablemente no la utilizará en la práctica.

Nuestra `SimpleRnn` tiene un par de características que no gustan demasiado. Una es que se utiliza su estado oculto entero para actualizar la entrada cada vez que se le llama, y la otra que el estado se sobrescribe por completo cada vez que se le llama. Ambas dificultan el entrenamiento; en particular, complican que el modelo aprenda dependencias de largo alcance.

Por esta razón, casi nadie utiliza este tipo de RNN sencilla. Lo que hace la gente es emplear variantes más complicadas como las redes LSTM (*Long Short-Term Memory*: memoria a largo y corto plazo) o GRU (*Gated Recurrent Unit*: unidad recurrente cerrada), que tienen muchos más parámetros y utilizan “puertas” parametrizadas que solo permiten que parte del estado se actualice (y solo parte del estado que se va a utilizar) en cada paso temporal. No hay nada especialmente difícil en estas variantes; sin embargo, implican mucho más código, cuya lectura no sería (en mi opinión) lógicamente más edificante. El código de este capítulo que puede conseguirse en GitHub[2] incluye una implementación LSTM. Le animo a que lo compruebe, pero es un poco tedioso, por lo que no lo discutiremos aquí con más detalle.

Otra particularidad de nuestra implementación es que da solo un “paso” cada vez y nos obliga a reiniciar manualmente el estado oculto. Una implementación de la RNN más práctica podría aceptar secuencias de entradas, poner su estado oculto a ceros al principio de cada secuencia, y producir secuencias de salidas. Las nuestras podrían sin duda ser modificadas para que se comporten de este modo, pero esto requeriría de nuevo más

código y complejidad para acabar consiguiendo poco conocimiento adicional.

Ejemplo: utilizar una RNN a nivel de carácter

Al nuevo vicepresidente recién contratado de Estrategia de Marcas no se le ocurrió el nombre de DataSciencester, y (en consecuencia) sospecha que un nombre mejor podría suponer un mayor éxito para la compañía. Por esta razón le pide que use ciencia de datos para sugerir posibles nombres nuevos.

Una aplicación "ingeniosa" de las redes RNN es emplear caracteres (en lugar de palabras) como entradas, entrenarlos para que aprendan los sutiles patrones de lenguaje de un conjunto de datos cualquiera y utilizarlos después para generar instancias ficticias a partir de ese conjunto de datos.

Por ejemplo, se podría adiestrar una RNN para que aprenda nombres de grupos de música alternativa, utilizar el modelo entrenado para generar nombres nuevos de grupos falsos, y después elegir a mano los más divertidos y compartirlos en Twitter. ¡Risas aseguradas!

Aún habiendo siendo testigo de este truco las veces suficientes como para no considerarlo ya inteligente, decide darle otra oportunidad.

Tras investigar un poco, descubre que el acelerador de *startups* Y Combinator ha publicado una lista de sus 100 (en realidad 101) *startups* de más éxito[3], que parece un buen punto de partida. Revisando la página, observa que los nombres de empresas residen todos en etiquetas `<b class="h4">`, lo que significa que es fácil aplicar sus habilidades de raspado web para recuperarlos:

```
from bs4 import BeautifulSoup
import requests
url = "https://www.ycombinator.com/topcompanies/"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
# Tenemos las empresas dos veces, así usa comprensión de conjunto para
deduplicar.
companies = list({b.text
             for b in soup("b")
             if "h4" in b.get("class", ())})
assert len(companies) == 101
```

Como siempre, la página puede cambiar (o desaparecer), en cuyo caso este código no funcionará. Si ocurre esto, puede usar sus habilidades de ciencia de datos recién adquiridas para arreglarlo o simplemente obtener la lista del sitio web de GitHub.

Así que ¿cuál es nuestro plan? Entrenaremos un modelo para predecir el siguiente carácter de un nombre, dado el actual y un estado oculto que representa todos los caracteres que hemos visto hasta ahora.

Como es habitual, predeciremos realmente una distribución de probabilidad según los caracteres y entrenaremos nuestro modelo para que minimice la pérdida `SoftmaxCrossEntropy`.

Una vez adiestrado, podemos usar el modelo para generar algunas probabilidades, extraer aleatoriamente un carácter según esas probabilidades y después pasar ese carácter como su siguiente entrada. Ello nos permitirá generar nombres de empresas utilizando los pesos aprendidos.

Para empezar, deberíamos construir un `Vocabulary` partiendo de los caracteres de los nombres:

```
vocab = Vocabulary([c for company in companies for c in company])
```

Además, utilizaremos elementos especiales que marquen el inicio y fin de un nombre de empresa, lo que permite al modelo aprender cuáles son los caracteres por los que debe empezar un nombre de empresa y, asimismo, saber averiguar cuándo ha acabado un nombre de empresa.

Utilizaremos los caracteres regex para el inicio y el final, que (con suerte) no aparecen en nuestra lista de empresas:

```
START = "^"
STOP = "$"

# Tenemos que añadirlos también al vocabulario.
vocab.add(START)
vocab.add(STOP)
```

Para nuestro modelo, aplicaremos one-hot-encode a cada carácter, lo pasaremos por dos `SimpleRnn` y usaremos después una capa `Linear` para generar los marcadores para cada siguiente carácter posible:

```
HIDDEN_DIM = 32   # Experimente con distintos tamaños.
rnn1 = SimpleRnn(input_dim=vocab.size, hidden_dim=HIDDEN_DIM)
```

```
rnn2 = SimpleRnn(input_dim=HIDDEN_DIM, hidden_dim=HIDDEN_DIM)
linear = Linear(input_dim=HIDDEN_DIM, output_dim=vocab.size)
model = Sequential([
  rnn1,
  rnn2,
  linear
])
```

Imaginemos en este momento que hemos entrenado este modelo. Escribamos la función que lo utiliza para generar nuevos nombres de empresas, utilizando la función `sample_from` del apartado anterior "Modelado de temas".

```
from scratch.deep_learning import softmax
def generate(seed: str = START, max_len: int = 50) -> str:
  rnn1.reset_hidden_state()     # Reinicia ambos estados ocultos
  rnn2.reset_hidden_state()
  output = [seed]               # Inicia la salida con el seed especificado
  # Sigue hasta producir el carácter STOP o alcanzar la longitud máxima
  while output[-1] != STOP and len(output) < max_len:
    # Usa el último carácter como entrada
    input = vocab.one_hot_encode(output[-1])
    # Genera marcadores usando el modelo
    predicted = model.forward(input)
    # Los convierte en probabilidades y dibuja un char_id aleatorio
    probabilities = softmax(predicted)
    next_char_id = sample_from(probabilities)
    # Añade el carácter correspondiente a nuestra salida
    output.append(vocab.get_word(next_char_id))
  # Se deshace de los caracteres START y END y devuelve la palabra
  return ‘’.join(output[1:-1])
```

Por fin estamos preparados para entrenar nuestra RNN a nivel de carácter. ¡Tardará un poco!

```
loss = SoftmaxCrossEntropy()
optimizer = Momentum(learning_rate=0.01, momentum=0.9)
for epoch in range(300):
    random.shuffle(companies)           # Entrena en un orden distinto cada
                                        epoch.
    epoch_loss = 0                      # Controla la pérdida.
    for company in tqdm.tqdm(companies):
      rnn1.reset_hidden_state()         # Reinicia ambos estados ocultos.
```

```
      rnn2.reset_hidden_state()
      company = START + company +       # Añade caracteres START y STOP.
      STOP
   # El resto es nuestro bucle de entrenamiento habitual, salvo que las
   entradas
   # y el objetivo son los caracteres con one-hot-encode anterior y
   posterior.
   for prev, next in zip(company, company[1:]):
      input = vocab.one_hot_encode(prev)
      target = vocab.one_hot_encode(next)
      predicted = model.forward(input)
      epoch_loss += loss.loss(predicted, target)
      gradient = loss.gradient(predicted, target)
      model.backward(gradient)
      optimizer.step(model)
# Cada epoch imprime la pérdida y genera un nombre.
print(epoch, epoch_loss, generate())
# Baja el ritmo de aprendizaje para los últimos 100 epoch.
# No hay ninguna razón para esto, pero parece funcionar.
if epoch == 200:
   optimizer.lr *= 0.1
```

Después de entrenar, el modelo genera varios nombres reales de la lista (lo que no resulta sorprendente, ya que el modelo tiene bastante capacidad y no muchos datos de entrenamiento), además de nombres que solo son ligeramente distintos a los nombres de entrenamiento (Scripe, Loinbare, Pozium), nombres que parecen genuinamente creativos (Benuus, Cletpo, Equite, Vivest) y nombres que son bastante malos, pero aún así parecen palabras (SFitreasy, Sint ocanelp, GliyOx, Doorboronelhav).

Es una lástima que, como la mayoría de los resultados de RNN a nivel de carácter, solo sean medianamente inteligentes, así que el vicepresidente de Estrategia de Marcas termina viéndose incapaz de utilizarlos.

Si subimos la dimensión oculta a 64, obtenemos textualmente muchos más nombres de la lista; si lo bajamos a 8, obtenemos principalmente palabras sin sentido. El vocabulario y los pesos finales de todos estos tamaños de modelo están disponibles en el sitio de GitHub, y puede emplear `load_weights` y `load_vocab` para utilizarlos si así lo desea.

Como hemos dicho antes, el código de GitHub para este capítulo contiene también la implementación de una LSTM, que, si lo desea, puede intercambiar tranquilamente como sustituto de las funciones `SimpleRnn` de

nuestro modelo de nombres de empresas.

Para saber más

- NLTK, en http://www.nltk.org/, es una librería conocida de herramientas PLN para Python. Tiene su propio libro completo en http://www.nltk.org/book/, que está disponible en línea.
- gensim, en http://radimrehurek.com/gensim/, es una librería de Python para modelado de temas, desde luego una apuesta mejor que nuestro modelo desde cero.
- spaCy, en https://spacy.io/, es una librería para "procesamiento del lenguaje natural de alta potencia en Python", y es también bastante conocida.
- Andrej Karpathy tiene un famoso blog de título "*The Unreasonable Effectiveness of Recurrent Neural Networks*", en http://karpathy.github.io/2015/05/21/rnn-effectiveness/, que vale la pena leer.
- Mi trabajo diario consiste en crear AllenNLP, en https://allennlp.org/, una librería de Python para investigación PLN (al menos, cuando este libro fue publicado, era así). La librería queda fuera del alcance de este libro, pero quizá le resulte interesante, y tiene además una demo interactiva de muchos modelos de PLN de vanguardia.

---

[1] http://oreil.ly/1Cd6ykN.

[2] https://github.com/joelgrus/datascience-from-scratch.

[3] https://www.ycombinator.com/topcompanies.

22 Análisis de redes

Las conexiones con todas las cosas que te rodean definen literalmente quién eres.

—Aaron O'Connell

Muchos problemas de datos interesantes pueden plantearse de manera fructífera en términos de redes, que consisten en nodos de algún tipo y las aristas o bordes que los conectan.

Los amigos de Facebook, por ejemplo, forman los nodos de una red, cuyas aristas son las relaciones de amistad. La propia World Wide Web es un ejemplo menos obvio, siendo cada página web un nodo y cada hiperenlace de una página a otra una arista.

La amistad de Facebook es mutua; si yo soy su amigo de Facebook, entonces usted tiene que ser necesariamente mi amigo. En este caso, decimos que las aristas son indirectas. Los hiperenlaces no lo son (mi sitio web conecta con `whitehouse.gov`, pero, por razones que me resultan inexplicables, la página del gobierno se niega a conectar con mi sitio). Este tipo de aristas se denominan directas. Veremos ambos tipos de redes.

Centralidad de intermediación

En el capítulo 1 escribimos código para obtener los conectores clave de la red DataSciencester contando el número de amigos que tenía cada usuario. Ahora disponemos de herramientas suficientes para estudiar otras estrategias. Emplearemos la misma red, pero ahora vamos a utilizar clases `NamedTuple` para los datos.

Recordemos que la red (figura 22.1) comprendía usuarios:

```
from typing import NamedTuple
class User(NamedTuple):
   id: int
```

```
   name: str
users = [User(0, "Hero"), User(1, "Dunn"), User(2, "Sue"), User(3, "Chi"),
 User(4, "Thor"), User(5, "Clive"), User(6, "Hicks"),
 User(7, "Devin"), User(8, "Kate"), User(9, "Klein")]
```

Y amistades:

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

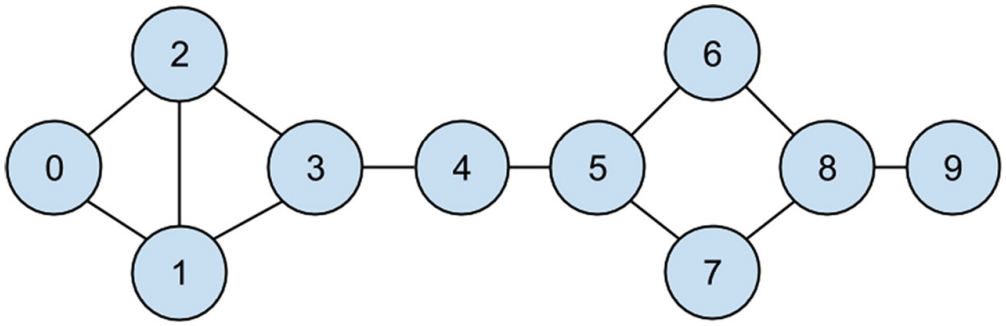


Figura 22.1. La red DataSciencester.

Será más fácil trabajar con las amistades como un `dict`:

```
from typing import Dict, List
# escribe alias para controlar Friendships
Friendships = Dict[int, List[int]]
friendships: Friendships = {user.id: [] for user in users}
for i, j in friend_pairs:
   friendships[i].append(j)
   friendships[j].append(i)
assert friendships[4] == [3, 5]
assert friendships[8] == [6, 7, 9]
```

Cuando terminamos en aquel momento, no estábamos satisfechos con nuestra noción de centralidad de grado, que en realidad no estaba de acuerdo con nuestra intuición sobre quiénes eran los conectores clave de la red.

Una métrica alternativa es la centralidad de intermediación, que identifica personas que suelen estar en las rutas más cortas entre pares de otras personas. En particular, la centralidad de intermediación del nodo *i* se calcula sumando, por cada par de nodos *j* y *k*, la proporción de caminos más cortos

entre el nodo *j* y el nodo *k* que pasan por *i*.

Es decir, para averiguar la centralidad de intermediación de Thor, tendremos que calcular todos los caminos más cortos entre todos los pares de personas que no son Thor. Después, tendremos que contar cuántos de estos caminos pasan por Thor. Por ejemplo, el único camino más corto entre Chi (id 3) y Clive (id 5) pasa por Thor, mientras que ninguna de las dos rutas más cortas entre Hero (id 0) y Chi (id 3) pasa por él.

Así, como primer paso, tendremos que averiguar las rutas más cortas entre todos los pares de personas. Existen varios algoritmos bastante sofisticados para hacer esto de una forma eficiente, pero (como es casi siempre el caso) utilizaremos un algoritmo menos eficiente y más fácil de comprender.

Este algoritmo (una implementación de la búsqueda en anchura) es uno de los más complicados del libro, así que hablaremos de él detenidamente:

1. Nuestro objetivo es una función que toma un valor `from_user` y encuentra todos los caminos más cortos al resto de los usuarios.
2. Representaremos cada ruta como una `list` de identificadores de usuario. Como todos los caminos empiezan en `from_user`, no incluiremos su identificador en la lista, lo que significa que la longitud de la lista que representa la ruta será la longitud de la propia ruta.
3. Mantendremos un diccionario llamado `shortest_paths_to`, donde las claves son identificadores de usuario y los valores son listas de rutas que terminan en el usuario con el ID especificado. Si hay un único camino más corto, la lista solo lo contendrá a él. Si hay varias rutas más cortas, la lista las contendrá todas.
4. También tendremos una cola denominada `frontier`, que contiene los usuarios que queremos explorar en el orden en el que queremos hacerlo. Los almacenaremos como pares `(prev_user, user)`, de modo que sepamos cómo llegamos a cada uno. Inicializaremos la cola con todos los vecinos de `from_user` (no hemos hablado de las colas, que son estructuras de datos optimizadas para operaciones “añadir al final” y “eliminar del frente”. En Python, se implementan como `collections.deque`, que es una cola de doble final).

5. Cuando exploremos el grafo, siempre que encontremos nuevos vecinos que no conozcan ya los caminos más cortos, los añadiremos al final de la cola para explorarlos después, con el usuario actual como `prev_user`.
6. Cuando sacamos a un usuario de la cola y nunca lo hemos encontrado antes, hemos hallado sin duda una o más rutas más cortas hasta él (cada una de las rutas más cortas a `prev_user` con un paso más añadido).
7. Cuando sacamos a un usuario de la cola, pero sí lo hemos encontrado antes, entonces o bien hemos encontrado otra ruta más corta (en cuyo caso deberíamos añadirla) o una más larga (en cuyo caso no deberíamos añadirla).
8. Cuando no quedan más usuarios en la cola, hemos explorado todo el grafo (o, al menos, las partes de él a las que se puede llegar desde el usuario inicial) y hemos terminado.

Podemos combinar todo esto en una función (bastante grande):

```
from collections import deque
Path = List[int]
def shortest_paths_from(from_user_id: int,
                  friendships: Friendships) -> Dict[int, List[Path]]:
  # Un diccionario de user_id a *todas* las rutas más cortas a ese usuario.
  shortest_paths_to: Dict[int, List[Path]] = {from_user_id: [[]]}
  # Una cola de (previous user, next user) que debemos revisar.
  # Empieza con todos los pares (from_user, friend_of_from_user).
  frontier = deque((from_user_id, friend_id)
              for friend_id in friendships[from_user_id])
  # Sigue hasta vaciar la cola.
  while frontier:
    # Elimina el par que va siguiente en la cola.
    prev_user_id, user_id = frontier.popleft()
    # Debido al modo en que estamos añadiendo a la cola,
    # ya sabemos sin duda algunas rutas más cortas a prev_user.
    paths_to_prev_user = shortest_paths_to[prev_user_id]
    new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]
    # Quizá ya conocemos una ruta más corta a user_id.
    old_paths_to_user = shortest_paths_to.get(user_id, [])
    # ¿Cuál es la ruta más corta hasta aquí vista hasta ahora?
    if old_paths_to_user:
      min_path_length = len(old_paths_to_user[0])
```

```
      else:
        min_path_length = float('inf')
      # Solo guarda rutas que no son demasiado largas y nuevas.
      new_paths_to_user = [path
for path in new_paths_to_user
if len(path) <= min_path_length
and path not in old_paths_to_user]
      shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user
      # Añade vecinos nunca vistos a la frontera.
      frontier.extend((user_id, friend_id)
for friend_id in friendships[user_id]
if friend_id not in shortest_paths_to)
return shortest_paths_to
```

Ahora calculemos todos los caminos más cortos:

```
# Para cada from_user, para cada to_user, una lista de rutas más cortas.
shortest_paths = {user.id: shortest_paths_from(user.id, friendships)
             for user in users}
```

Finalmente, estamos listos para calcular la centralidad de intermediación. Por cada par de nodos *i* y *j*, conocemos las *n* rutas más cortas de *i* a *j*. Entonces, para cada una de esas rutas, simplemente sumamos 1/n a la centralidad de cada nodo en esa ruta:

```
betweenness_centrality = {user.id: 0.0 for user in users}

for source in users:
  for target_id, paths in shortest_paths[source.id].items():
    if source.id < target_id:      # no cuenta doble
      num_paths = len(paths) # ¿cuántas rutas más cortas?
      contrib = 1 / num_paths      # contribución a la centralidad
      for path in paths:
        for between_id in path:
          if between_id not in [source.id, target_id]:
            betweenness_centrality[between_id] += contrib
```

Como se muestra en la figura 22.2, los usuarios 0 y 9 tienen centralidad 0

(ya que ninguno está en ninguno de los caminos más cortos entre otros usuarios), mientras que 3, 4 y 5 tienen todos centralidades elevadas (ya que los tres residen en muchas de las rutas más cortas).

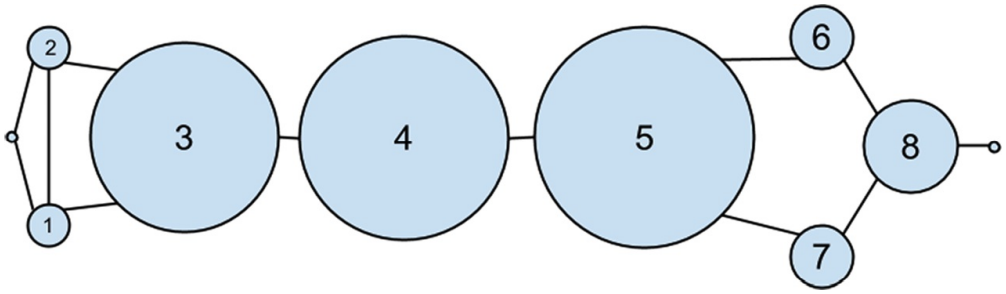


Figura 22.2. La red DataSciencester dimensionada por la centralidad de intermediación.

> **Nota:** Por lo general, los números de centralidad no son tan significativos por sí mismos. Lo que nos interesa es cómo se comparan los números de cada nodo con los de otros nodos.

Otra medida que podemos revisar es la centralidad de cercanía. Primero, para cada usuario calculamos su lejanía, que es la suma de las longitudes de sus caminos más cortos hacia el resto de los usuarios. Como ya hemos calculado las rutas más cortas entre cada par de nodos, es fácil añadir sus longitudes (si hay varios caminos más cortos, tienen todos la misma longitud, de modo que simplemente podemos mirar el primero).

```
def farness(user_id: int) -> float:
   """the sum of the lengths of the shortest paths to each other user"""
   return sum(len(paths[0])
           for paths in shortest_paths[user_id].values())
```

Tras de lo cual cuesta muy poco calcular la centralidad de cercanía (figura 22.3):

```
closeness_centrality = {user.id: 1 / farness(user.id) for user in users}
```

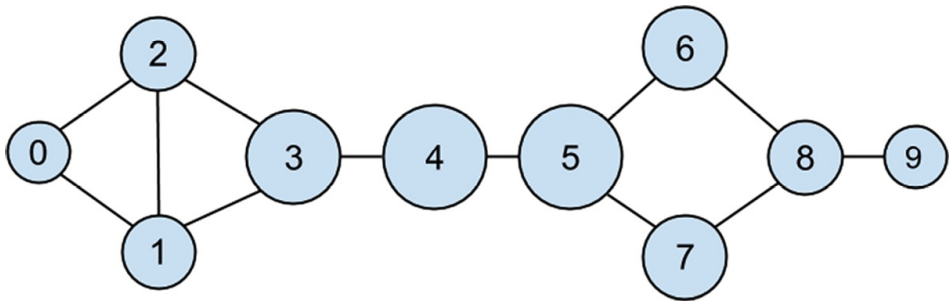


Figura 22.3. La red DataSciencester dimensionada por la centralidad de cercanía.

Aquí hay mucha menos variación (incluso los nodos muy centrales siguen estando bastante lejos de los nodos que están lejos en la periferia).

Como ya hemos visto, calcular rutas más cortas es un poco complicado. Por esta razón, la centralidad de intermediación y cercanía no se utilizan demasiado en redes grandes, pero sí se utiliza más a menudo la centralidad de vector propio, menos intuitiva (pero en general más fácil de calcular).

Centralidad de vector propio

Para hablar sobre la centralidad de vector propio, tenemos que hablar de los vectores propios y, para hablar de los vectores propios, tenemos que hablar de la multiplicación de matrices.

Multiplicación de matrices

Si A es una matriz n x m y B es otra m x k (obsérvese que la segunda dimensión de A es la misma que la primera dimensión de B), entonces su producto AB es la matriz n x k cuya entrada (i,j)-ésima es:

$$A_{i1}B_{1j} + A_{i2}B_{2j} + ... + A_{im}B_{mj}$$

Que es precisamente el producto punto de la fila i-ésima de A (pensada como un vector) por la columna j-ésima de B (también pensada como un vector).

Podemos implementar esto utilizando la función `make_matrix` del capítulo

4:

```
from scratch.linear_algebra import Matrix, make_matrix, shape
def matrix_times_matrix(m1: Matrix, m2: Matrix) -> Matrix:
   nr1, nc1 = shape(m1)
   nr2, nc2 = shape(m2)
   assert nc1 == nr2, "must have (# of columns in m1) == (# of rows in m2)"
   def entry_fn(i: int, j: int) -> float:
      """dot product of i-th row of m1 with j-th column of m2"""
      return sum(m1[i][k] * m2[k][j] for k in range(nc1))
   return make_matrix(nr1, nc2, entry_fn)
```

Si pensamos en un vector de *m* dimensiones como en una matriz `(m, 1)`, podemos multiplicarlo por una matriz `(n, m)` para obtener otra matriz `(n, 1)`, que podemos considerar como un vector de *n* dimensiones.

Esto significa que otra forma de pensar en una matriz `(n, m)` es como si fuera un mapeado lineal que transforma vectores de *m* dimensiones en vectores de *n* dimensiones:

```
from scratch.linear_algebra import Vector, dot
def matrix_times_vector(m: Matrix, v: Vector) -> Vector:
   nr, nc = shape(m)
   n = len(v)
   assert nc == n, "must have (# of cols in m) == (# of elements in v)"
   return [dot(row, v) for row in m]     # el resultado tiene longitud nr
```

Cuando *A* es una matriz cuadrada, esta operación mapea vectores de *n* dimensiones a otros vectores de *n* dimensiones. Es posible que, para una cierta matriz *A* y un determinado vector *v*, cuando *A* trabaja sobre *v* obtenemos de vuelta un escalar múltiplo de *v* (es decir, que el resultado es un vector que apunta en la misma dirección que *v*). Cuando esto ocurre (y cuando, además, *v* no es un vector todo ceros), denominamos a *v* vector propio de *A*, y al multiplicador lo denominamos valor propio.

Una posible forma de encontrar un vector propio de *A* es eligiendo un vector inicial *v*, aplicando `matrix_times_vector`, redimensionando el resultado para que tenga magnitud 1, y repitiendo hasta que el proceso converja:

```
from typing import Tuple
import random
from scratch.linear_algebra import magnitude, distance
def find_eigenvector(m: Matrix,
                tolerance: float = 0.00001) -> Tuple[Vector, float]:
   guess = [random.random() for _ in m]
   while True:
      result = matrix_times_vector(m, guess)     # transforma guess
      norm = magnitude(result)                   # calcula norm
      next_guess = [x / norm for x in result]    # redimensiona
      if distance(guess, next_guess) < tolerance:
         # convergencia, así devuelve (vector propio, valor propio)
         return next_guess, norm
      guess = next_guess
```

Por construcción, el valor `guess` devuelto es un vector tal que, cuando se le aplica `matrix_times_vector` y se redimensiona para que tenga longitud 1, se obtiene de vuelta un vector muy próximo a sí mismo (lo que significa que es un vector propio).

No todas las matrices de números reales tienen vectores propios y valores propios. Por ejemplo, la matriz:

```
rotate = [[ 0, 1],
 [-1, 0]]
```

Gira los vectores 90 grados en el sentido de las agujas del reloj, lo que significa que el único vector que convierte en un escalar múltiplo de sí mismo es un vector de ceros. Si se intentara `find_eigenvector(rotate)`, funcionaría para siempre. Incluso las matrices que tienen vectores propios pueden quedarse a veces atascadas en ciclos. Veamos la matriz:

```
flip = [[0, 1],
     [1, 0]]
```

Esta matriz convierte cualquier vector `[x, y]` en `[y, x]`. Esto significa que, por ejemplo, `[1, 1]` es un vector propio con valor propio 1. Sin embargo, si se empieza con un vector aleatorio con coordenadas desiguales, lo que hará `find_eigenvector` será intercambiar repetidamente las

coordenadas para siempre (las librerías más avanzadas como NumPy usan distintos métodos que funcionarían en este caso). No obstante, cuando `find_eigenvector` devuelve un resultado, dicho resultado es de hecho un vector propio.

Centralidad

¿Cómo ayuda esto a comprender la red DataSciencester? Para empezar, tendremos que representar las conexiones de nuestra red como una `adjacency_matrix`, cuya entrada (*i*,*j*)-ésima es o bien 1 (si los usuarios *i* y *j* son amigos) o 0 (si no lo son):

```
def entry_fn(i: int, j: int):
   return 1 if (i, j) in friend_pairs or (j, i) in friend_pairs else 0
n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

La centralidad de vector propio para cada usuario es entonces la entrada correspondiente a ese usuario en el vector propio devuelto por `find_eigenvector` (figura 22.4).

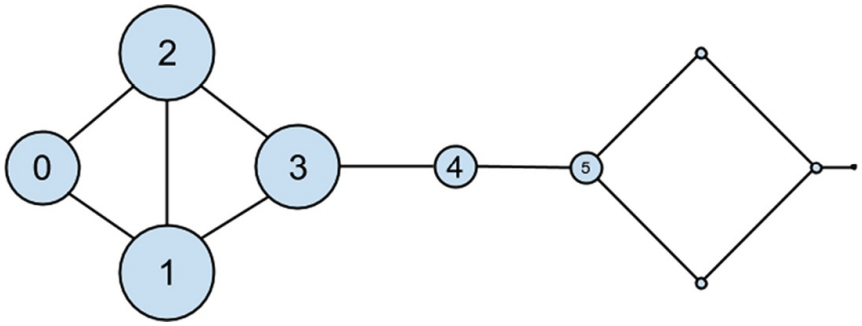


Figura 22.4. La red DataSciencester dimensionada por la centralidad de vector propio.

Nota: Por razones técnicas que quedan fuera del alcance de este libro, cualquier matriz de adyacencia no nula tiene necesariamente un vector propio, cuyos valores son todos no negativos. Afortunadamente para nosotros, para esta `adjacency_matrix`, nuestra función `find_eigenvector` lo localiza.

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

Los usuarios con elevada centralidad de vector propio deberían ser los que tienen muchas conexiones, y con gente que tiene de por sí también una alta centralidad.

Aquí los usuarios 1 y 2 son los más centrales, ya que ambos tienen tres conexiones con personas que son también muy centrales. A medida que nos alejamos de ellos, las centralidades de las personas caen de manera constante.

En una red tan pequeña, la centralidad de vector propio se comporta de un modo algo errático. Si intentamos sumar o restar enlaces, descubrimos que pequeños cambios en la red pueden modificar de forma drástica los números de centralidad. En una red mucho más grande, este no sería el caso.

Aún no hemos explicado por qué un vector propio podría ofrecer una noción razonable de centralidad. Ser un vector propio significa que si se calcula:

```
matrix_times_vector(adjacency_matrix, eigenvector_centralities)
```

El resultado es un escalar múltiplo de `eigenvector_centralities`.

Si observamos cómo funciona la multiplicación de matrices, vemos que `matrix_times_vector` produce un vector cuyo *i*-ésimo elemento es:

```
dot(adjacency_matrix[i], eigenvector_centralities)
```

Que es precisamente la suma de las centralidades de vector propio de los usuarios conectados con el usuario *i*.

En otras palabras, las centralidades de vector propio son números, uno por usuario, tales que el valor de cada usuario es una constante múltiplo de la suma de los valores de sus vecinos. En este caso, la centralidad significa estar conectado a personas que a su vez son centrales. Cuanto mayor sea la centralidad a la que se está conectado, más central se es. Por supuesto, esto es una definición circular (los vectores propios son la forma de romper esa circularidad).

Otra forma de entender esto es pensando en lo que está haciendo aquí `find_eigenvector`. Empieza asignando a cada nodo una centralidad aleatoria. Después repite los dos pasos siguientes hasta que el proceso converge:

1. Da a cada nodo una nueva puntuación de centralidad, que es igual a la

suma de las puntuaciones de centralidad (antiguas) de sus vecinos.
2. Redimensiona el vector de centralidades para que tenga magnitud 1.

Aunque las matemáticas que subyacen en este cálculo puedan parecer un poco opacas al principio, el cálculo en sí es relativamente sencillo (a diferencia, digamos, de la centralidad de intermediación) y es bastante fácil de aplicar en grafos aún más grandes (al menos, si se utiliza una librería de álgebra lineal de verdad, es fácil de aplicar en grafos grandes. Utilizando nuestra implementación de matrices como listas, costaría bastante).

Grafos dirigidos y PageRank

DataSciencester no está ganando mucho terreno, de modo que el vicepresidente de Ingresos considera cambiar de un modelo de amistades a otro de recomendaciones o respaldos. Resulta que a nadie en particular le interesa qué científicos de datos son amigos, pero a los reclutadores tecnológicos sí les importa qué científicos de datos son respetados por sus iguales. En este nuevo modelo, detectaremos las recomendaciones `(source, target)` que ya no representan una relación recíproca, sino más bien que `source` recomiende a `target` como un fabuloso científico de datos (figura 22.5).

Tendremos que tener en cuenta esta asimetría:

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

Tras de lo cual podemos encontrar fácilmente los científicos de datos `most_endorsed` y vender esa información a los reclutadores:

```
from collections import Counter

endorsement_counts = Counter(target for source, target in endorsements)
```

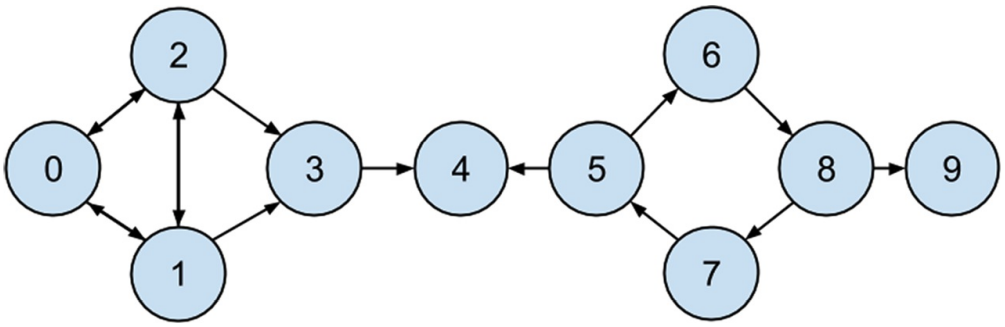


Figura 22.5. La red DataSciencester de recomendaciones.

Sin embargo, “número de recomendaciones” es una métrica fácil con la que jugar. Todo lo que hay que hacer es crear cuentas falsas y lograr recomendaciones; o quedar con los amigos para respaldarse mutuamente (como parecen haber hecho los usuarios 0, 1 y 2). Una mejor métrica tendría en cuenta quién le recomienda. Las recomendaciones recibidas de personas que son a su vez muy respaldadas por otros deberían contar más de algún modo que las recomendaciones de gente que tiene pocos seguidores. Esta es la esencia del algoritmo PageRank, empleado por Google para ordenar sitios web basándose en qué otras páginas web enlazan con ellos, qué otros sitios enlazan con esos, y así sucesivamente.

(Si esto le recuerda de alguna forma a la centralidad de vector propio, ¡premio!).

Una versión simplificada es algo así:

1. Hay un total de 1.0 (o 100 %) PageRank en la red.
2. Inicialmente este PageRank está distribuido por igual entre nodos.
3. En cada paso, una gran fracción del PageRank de cada nodo está distribuido por igual entre sus enlaces de salida.
4. En cada paso, el resto del PageRank de cada nodo está distribuido por igual entre todos los nodos.

```
import tqdm
def page_rank(users: List[User],
              endorsements: List[Tuple[int, int]],
              damping: float = 0.85,
              num_iters: int = 100) -> Dict[int, float]:
  # Calcula cuánta gente recomienda a cada persona
  outgoing_counts = Counter(target for source, target in endorsements)
```

```
    # Inicialmente distribuye PageRank por igual
    num_users = len(users)
    pr = {user.id : 1 / num_users for user in users}
    # Pequeña fracción de PageRank que obtiene cada nodo en cada iteración
    base_pr = (1–damping) / num_users
    for iter in tqdm.trange(num_iters):
       next_pr = {user.id : base_pr for user in            # start with
       users}                                              base_pr
       for source, target in endorsements:
          # Suma fracción amortiguada de source pr a target
          next_pr[target] += damping * pr[source] / outgoing_counts[source]
       pr = next_pr
    return pr
```

Si averiguamos el orden de las páginas:

```
pr = page_rank(users, endorsements)
# Thor (user_id 4) tiene la posición de página más alta
assert pr[4] > max(page_rank
            for user_id, page_rank in pr.items()
            if user_id != 4)
```

PageRank (figura 22.6) identifica al usuario 4 (Thor) como el científico de datos mejor posicionado.

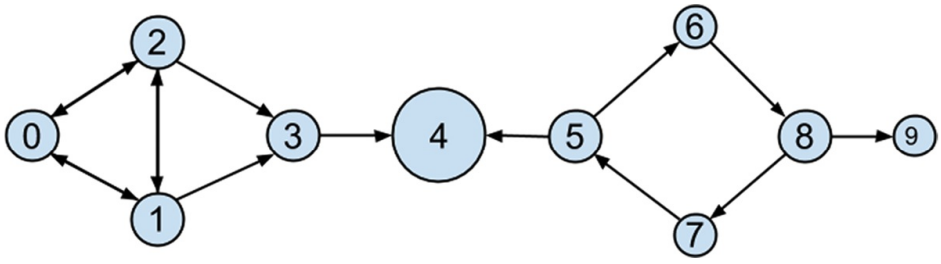


Figura 22.6. La red DataSciencester dimensionada por PageRank.

Aunque Thor tenga menos recomendaciones (dos) que los usuarios 0, 1 y 2, sus seguidores traen consigo buenas posiciones de sus propias recomendaciones. Además, los dos usuarios que le respaldan solo le apoyan a él, lo que significa que no tiene que dividir su posición con nadie más.

Para saber más

- Hay muchas otras nociones de centralidad, en https://es.wikipedia.org/wiki/Centralidad, además de las que hemos utilizado (aunque estas son las más conocidas).
- NetworkX (http://networkx.github.io/) es una librería Python para análisis de redes, que tiene funciones para calcular centralidades y visualizar grafos.
- Gephi, en https://gephi.org/, es una herramienta de visualización de redes basada en GUI que produce entre sus usuarios un sentimiento de amor-odio.

23 Sistemas recomendadores

¡Oh, naturaleza, naturaleza, por qué eres tan deshonesta, como para siempre enviar hombres con estas recomendaciones falsas al mundo!

—Henry Fielding

Otro problema de datos habitual es producir recomendaciones de artículos o productos. Por ejemplo, Netflix recomienda películas que a sus usuarios quizá les pueda interesar ver; Amazon recomienda productos que a sus usuarios quizá les pueda interesar comprar. En este capítulo veremos varias formas de utilizar datos para hacer este tipo de recomendaciones. En nuestro caso veremos el conjunto de datos `users_interests` que ya hemos utilizado antes:

```
users_interests = [
   ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
   ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
   ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
   ["R", "Python", "statistics", "regression", "probability"],
   ["machine learning", "regression", "decision trees", "libsvm"],
   ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
   ["statistics", "probability", "mathematics", "theory"],
   ["machine learning", "scikit-learn", "Mahout", "neural networks"],
   ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
   ["Hadoop", "Java", "MapReduce", "Big Data"],
   ["statistics", "R", "statsmodels"],
   ["C++", "deep learning", "artificial intelligence", "probability"],
   ["pandas", "R", "Python"],
   ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
   ["libsvm", "regression", "support vector machines"]
]
```

También abordaremos el problema de recomendar intereses nuevos a un usuario según sus intereses actuales.

Método manual

Antes de que existiera Internet, si uno necesitaba recomendaciones de libros, visitaba la biblioteca, donde el bibliotecario sugería de buen grado libros adaptados a los intereses solicitados u otros libros similares que pudieran gustar.

Dado el limitado número de usuarios e intereses de DataSciencester, le resultaría muy fácil pasarse toda una tarde recomendando manualmente intereses a cada usuario. Pero este método no amplía realmente bien, ya que se ve limitado por sus conocimientos y su imaginación (no estoy sugiriendo con esto que sus conocimientos e imaginación sean limitados). De modo que pensemos lo que podemos hacer con datos.

Recomendar lo que es popular

Un planteamiento sencillo es simplemente recomendar lo que es popular:

```
from collections import Counter
popular_interests = Counter(interest
for user_interests in users_interests
for interest in user_interests)
```

Que queda de la siguiente manera:

```
[('Python', 4),
  ('R', 4),
  ('Java', 3),
  ('regression', 3),
  ('statistics', 3),
  ('probability', 3),
  # ...
]
```

Teniendo ya escrito el código para esto, podemos sencillamente sugerir a un usuario los intereses más populares en los que no esté ya interesado:

```
from typing import List, Tuple
def most_popular_new_interests(
     user_interests: List[str],
     max_results: int = 5) -> List[Tuple[str, int]]:
   suggestions = [(interest, frequency)
              for interest, frequency in popular_interests.most_common()
              if interest not in user_interests]
   return suggestions[:max_results]
```

Así, si usted fuera el usuario 1, con los intereses siguientes:

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

Entonces le recomendaríamos:

```
[('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

Si fuera el usuario 3, que ya está interesado en muchos de estos temas, obtendría en su lugar:

```
[('Java', 3), ('HBase', 3), ('Big Data', 3),
   ('neural networks', 2), ('Hadoop', 2)]
```

Por supuesto, "como mucha gente está interesada en Python, a usted también debería interesarle" no es el argumento de venta más convincente. Si alguien acaba de entrar por primera vez en su sitio y no sabemos nada de esa persona, es posiblemente lo mejor que podemos hacer. Veamos cómo podemos mejorarlo basando las recomendaciones de cada usuario en sus intereses ya existentes.

Filtrado colaborativo basado en usuarios

Una forma de tener en cuenta los intereses de un usuario es buscar usuarios que de alguna forma sean similares, y después sugerirle los temas en los que esos usuarios están interesados. Para hacer esto, necesitaremos una forma de medir lo parecidos que son dos usuarios, y aplicaremos para ello la similitud de coseno, que ya utilizamos en el capítulo 21 para medir lo parecidos que eran dos vectores de palabras.

Aplicaremos esto a los vectores de ceros y unos, representando cada vector `v` los intereses de un usuario. `v[i]` será 1 si el usuario especificó el interés *i*, y 0 en caso contrario. De acuerdo con esto, "usuarios parecidos" significará "usuarios cuyos vectores de intereses apuntan casi siempre en la misma dirección". Los usuarios con intereses idénticos tendrán una similitud 1, y los que no los tengan idénticos tendrán similitud 0. En otros casos, la similitud caerá entre medias, indicando los números más cercanos a 1 "muy parecido" y los más cercanos a 0 "no muy parecido".

Un buen punto de partida es recopilar los intereses conocidos y asignarles índices (de forma implícita). Podemos hacerlo utilizando una comprensión de conjunto para hallar los intereses únicos, y después ordenarlos en una lista. El primer interés de la lista resultante será el interés 0, etc.:

```
unique_interests = sorted({interest
 for user_interests in users_interests
 for interest in user_interests})
```

Esto nos da una lista que empieza así:

```
assert unique_interests[:6] == [
   'Big Data',
   'C++',
   'Cassandra',
   'HBase',
   'Hadoop',
   'Haskell',
   # ...
]
```

Ahora queremos producir un vector de "intereses" de ceros y unos para cada usuario. Solo tenemos que pasar repetidas veces por la lista `unique_interests`, sustituyendo un 1 si el usuario tiene cada interés y 0 si no lo tiene:

```
def make_user_interest_vector(user_interests: List[str]) -> List[int]:
   """
   Given a list of interests, produce a vector whose ith element is 1
   if unique_interests[i] is in the list, 0 otherwise
   """
```

```
   return [1 if interest in user_interests else 0
        for interest in unique_interests]
```

Ahora podemos crear una lista de vectores de intereses de usuario:

```
user_interest_vectors = [make_user_interest_vector(user_interests)
                  for user_interests in users_interests]
```

Ahora `user_interests_vectors[i][j]` es igual a 1 si el usuario `i` especificó el interés `j`, y 0 si no fue así.

Como tenemos un conjunto de datos pequeño, no es problema calcular las similitudes por pares entre todos nuestros usuarios:

```
from scratch.nlp import cosine_similarity
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
               for interest_vector_j in user_interest_vectors]
               for interest_vector_i in user_interest_vectors]
```

Tras de lo cual `user_similarities[i][j]` nos da las similitudes entre los usuarios `i` y `j`:

```
# Los usuarios 0 y 9 comparten intereses en Hadoop, Java y Big Data
assert 0.56 < user_similarities[0][9] < 0.58, "several shared interests"

# Los usuarios 0 y 8 comparten solo un interés: Big Data
assert 0.18 < user_similarities[0][8] < 0.20, "only one shared interest"
```

En particular, `user_similarities[i]` es el vector de las similitudes del usuario `i` con los demás usuarios. Podemos usar esto para escribir una función que halle los usuarios más parecidos a un usuario dado. Nos aseguraremos de no incluir al propio usuario ni a ningún otro con similitud cero. Además, ordenaremos los resultados del más al menos parecido:

```
def most_similar_users_to(user_id: int) -> List[Tuple[int,
float]]:
   pairs = [(other_user_id, similarity)                          # Halla otros
 for other_user_id, similarity in                                # usuarios
                                                                 con
             enumerate(user_similarities[user_id])               # similitud
        if user_id != other_user_id and similarity > 0]          # no cero.
   return sorted(pairs,                                          # Ordena los
```

```
            key=lambda pair: pair[-1],                          # más
                                                                similares
            reverse=True)                                       # primero.
```

Por ejemplo, si llamamos a `most_similar_users_to(0)`, obtenemos:

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

¿Cómo usamos esto para sugerirle nuevos intereses a un usuario? Por cada interés, podemos sumar las similitudes de usuario de los otros usuarios interesados en él:

```
from collections import defaultdict
def user_based_suggestions(user_id: int,
          include_current_interests: bool = False):
  # Suma las similitudes
  suggestions: Dict[str, float] = defaultdict(float)
  for other_user_id, similarity in most_similar_users_to(user_id):
     for interest in users_interests[other_user_id]:
        suggestions[interest] += similarity
  # Las convierte en lista ordenada
  suggestions = sorted(suggestions.items(),
                key=lambda pair:            # weight
                pair[-1],
                reverse=True)
  # Y (quizá) ya excluye intereses
  if include_current_interests:
     return suggestions
  else:
     return [(suggestion, weight)
          for suggestion, weight in suggestions
          if suggestion not in users_interests[user_id]]
```

Si llamamos a `user_based_suggestions(0)`, los primeros intereses sugeridos son:

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
```

```
('Postgres', 0.50709255283711),
('NoSQL', 0.3380617018914066),
('neural networks', 0.1889822365046136),
('deep learning', 0.1889822365046136),
('artificial intelligence', 0.1889822365046136),
#...
]
```

Parecen sugerencias bastante aceptables para alguien cuyos intereses declarados son "Big Data" y otros relacionados con bases de datos (los pesos no son intrínsecamente significativos; simplemente los utilizamos para ordenar).

Este planteamiento no funciona tan bien cuando el número de elementos se incrementa mucho. Recordemos la maldición de la dimensionalidad del capítulo 12 (en espacios vectoriales de grandes dimensiones la mayoría de los vectores están muy alejados unos de otros, y apuntan además en direcciones muy distintas). Es decir, cuando hay un gran número de intereses, los "usuarios más similares" a un usuario dado pueden no serlo en absoluto.

Imaginemos un sitio como Amazon.com, en el que he comprado cientos de artículos en las últimas dos décadas. Podríamos intentar identificar usuarios parecidos a mí basándonos en patrones de compra, pero lo más probable es que no haya nadie en el mundo que tenga un historial de compra remotamente parecido al mío. Quienquiera que sea mi comprador "más parecido", probablemente no se parece en nada a mí, y sus compras casi seguro que me parecerían recomendaciones de lo peor.

Filtrado colaborativo basado en artículos

Un método alternativo es calcular directamente las similitudes entre intereses. Podemos entonces generar sugerencias para cada usuario, agregando intereses que sean similares a los que ya tienen actualmente.

Para empezar, transponemos nuestra matriz de intereses de usuario, de forma que las filas correspondan con los intereses y las columnas con los usuarios:

```
interest_user_matrix = [[user_interest_vector[j]
                  for user_interest_vector in user_interest_vectors]
                  for j, _ in enumerate(unique_interests)]
```

¿Cómo queda esto? La fila `j` de `interest_user_matrix` es la columna `j` de `user_interest_matrix`. Es decir, tiene un 1 por cada usuario con ese interés y un 0 por cada usuario sin ese interés.

Por ejemplo, `unique_interest[0]` es Big Data, de modo que `user_interest_matrix[0]` es:

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

Porque los usuarios 0, 8 y 9 indicaron interés en Big Data.

Ahora podemos utilizar de nuevo la similitud de coseno. Si precisamente los mismos usuarios están interesados en dos temas, su similitud será 1. Si no hay dos usuarios que estén interesados en ambos temas, su similitud será 0:

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                  for user_vector_j in interest_user_matrix]
                  for user_vector_i in interest_user_matrix]
```

Por ejemplo, podemos encontrar los intereses más similares a Big Data (interés 0) utilizando:

```
def most_similar_interests_to(interest_id: int):
   similarities = interest_similarities[interest_id]
   pairs = [(unique_interests[other_interest_id], similarity)
         for other_interest_id, similarity in enumerate(similarities)
         if interest_id != other_interest_id and similarity > 0]
   return sorted(pairs,
              key=lambda pair: pair[-1],
              reverse=True)
```

Lo que sugiere los siguientes intereses similares:

```
[('Hadoop', 0.8164965809277261),
('Java', 0.6666666666666666),
('MapReduce', 0.5773502691896258),
('Spark', 0.5773502691896258),
('Storm', 0.5773502691896258),
('Cassandra', 0.4082482904638631),
```

```
('artificial intelligence', 0.4082482904638631),
('deep learning', 0.4082482904638631),
('neural networks', 0.4082482904638631),
('HBase', 0.3333333333333333)]
```

Ahora podemos crear recomendaciones para un usuario sumando las similitudes de los intereses que son similares al suyo:

```
def item_based_suggestions(user_id: int,
include_current_interests: bool = False):
  # Suma los intereses similares
  suggestions = defaultdict(float)
  user_interest_vector = user_interest_vectors[user_id]
  for interest_id, is_interested in enumerate(user_interest_vector):
    if is_interested == 1:
      similar_interests = most_similar_interests_to(interest_id)
      for interest, similarity in similar_interests:
        suggestions[interest] += similarity
  # Los ordena por peso
  suggestions = sorted(suggestions.items(),
                key=lambda pair: pair[-1],
                reverse=True)
  if include_current_interests:
    return suggestions
  else:
    return [(suggestion, weight)
         for suggestion, weight in suggestions
         if suggestion not in users_interests[user_id]]
```

Para el usuario 0, esto genera las siguientes recomendaciones (aparentemente razonables):

```
[('MapReduce', 1.861807319565799),
('Postgres', 1.3164965809277263),
('MongoDB', 1.3164965809277263),
('NoSQL', 1.2844570503761732),
('programming languages', 0.5773502691896258),
('MySQL', 0.5773502691896258),
('Haskell', 0.5773502691896258),
('databases', 0.5773502691896258),
('neural networks', 0.4082482904638631),
('deep learning', 0.4082482904638631),
('C++', 0.4082482904638631),
('artificial intelligence', 0.4082482904638631),
```

```
('Python', 0.2886751345948129),
('R', 0.2886751345948129)]
```

Factorización de matrices

Como hemos visto, podemos representar las preferencias de nuestros usuarios como una matriz `[num_users, num_items]` de ceros y unos, donde los unos representan los artículos que han gustado y los ceros los que no han gustado.

En ocasiones podríamos tener clasificaciones numéricas; por ejemplo, al escribir una valoración en Amazon se asigna al artículo una puntuación que va de 1 a 5 estrellas. Podríamos representarlas con números de una matriz `[num_users, num_items]` (ignorando por ahora el problema de qué hacer con los artículos no valorados).

En esta sección supondremos que tenemos estos datos de valoraciones e intentaremos estudiar un modelo que pueda predecir la valoración para un determinado usuario y artículo.

Una forma de abordar el problema es suponer que cada usuario tiene un cierto "tipo" de artículo latente, que se puede representar como un vector de números, y que cada artículo tiene un "tipo" similar, pero de usuario.

Si los tipos del usuario se representan como una matriz `[num_users, dim]`, y la transposición de los tipos del artículo se representa como una matriz `[dim, num_items]`, su producto es una matriz `[num_users, num_items]`. En consecuencia, una forma de crear un modelo como este es "factorizando" la matriz de preferencias en el producto de una matriz de usuario por una matriz de artículo.

(Posiblemente esta idea de tipos latentes le recuerde a las incrustaciones de palabras que desarrollamos en el capítulo 21. Quédese con esta idea).

En lugar de trabajar con nuestro conjunto de datos inventado de 10 usuarios, usaremos el conjunto de datos 100k de MovieLens, que contiene valoraciones de 0 a 5 de muchas películas de muchos usuarios. Lo emplearemos para intentar construir un sistema que pueda predecir la valoración de cualquier par (usuario, película). Lo entrenaremos para que

prediga bien las películas que cada usuario ha valorado; con suerte generalizará entonces a las películas que el usuario no ha valorado.

Para empezar, adquiramos el conjunto de datos. Puede descargarlo en https://grouplens.org/datasets/movielens/100k/.

Descomprima y extraiga los archivos; solo utilizaremos dos de ellos:

```
# Apunta al directorio actual, cámbielo a donde estén sus archivos.
MOVIES = "u.item"      # pipe-delimited: movie_id|title|...
RATINGS = "u.data"     # tab-delimited: user_id, movie_id, rating, timestamp
```

Como solemos hacer, introducimos una `NamedTuple` para facilitarnos el trabajo:

```
from typing import NamedTuple
class Rating(NamedTuple):
   user_id: str
   movie_id: str
   rating: float
```

Nota: El identificador de la película y los identificadores de usuario son realmente números enteros, pero no son consecutivos, lo que significa que si trabajáramos con ellos así terminaríamos con un montón de dimensiones desaprovechadas (a menos que lo renumeráramos todo). De modo que, para simplificar las cosas, los trataremos como cadenas de texto.

Leamos ahora los datos y explorémoslos. El archivo de películas está delimitado por barras verticales (caracteres *pipe*) y tiene muchas columnas. Solo nos interesan las dos primeras, que son el identificador y el título:

```
import csv
# Especificamos esta codificación para evitar un UnicodeDecodeError.
# Ver: https://stackoverflow.com/a/53136168/1076346.
with open(MOVIES, encoding="iso-8859-1") as f:
   reader = csv.reader(f, delimiter="|")
   movies = {movie_id: title for movie_id, title, *_ in reader}
```

El archivo de valoraciones está delimitado por tabuladores y contiene cuatro columnas para `user_id`, `movie_id`, `rating` (de 1 a 5) y `timestamp`.

Ignoraremos la última, ya que no la necesitamos:

```
# Crea una lista de [Rating]
with open(RATINGS, encoding="iso-8859-1") as f:
   reader = csv.reader(f, delimiter="\t")
   ratings = [Rating(user_id, movie_id, float(rating))
           for user_id, movie_id, rating, _ in reader]
# 1682 películas valoradas por 943 usuarios
assert len(movies) == 1682
assert len(list({rating.user_id for rating in ratings})) == 943
```

Con estos datos se pueden hacer muchos análisis de exploración interesantes; nos podrían interesar, por ejemplo, las valoraciones medias de las películas de *Star Wars* (el conjunto de datos es de 1998, lo que significa que todavía le falta un año a *La Amenaza Fantasma* para estrenarse):

```
import re
# Estructura de datos para acumular valoraciones por movie_id
star_wars_ratings = {movie_id: []
                for movie_id, title in movies.items()
                if re.search("Star Wars|Empire Strikes|Jedi", title)}
# Itera por las valoraciones, acumulando las de Star Wars
for rating in ratings:
   if rating.movie_id in star_wars_ratings:
      star_wars_ratings[rating.movie_id].append(rating.rating)
# Calcula la valoración media para cada película
avg_ratings = [(sum(title_ratings) / len(title_ratings), movie_id)
           for movie_id, title_ratings in star_wars_ratings.items()]
# Y las muestra en pantalla en orden
for avg_rating, movie_id in sorted(avg_ratings, reverse=True):
   print(f"{avg_rating:.2f} {movies[movie_id]}")
```

Todas están bastante bien valoradas:

```
4.36 Star Wars (1977)
4.20 Empire Strikes Back, The (1980)
4.01 Return of the Jedi (1983)
```

Así que probemos a dar con un modelo que prediga estas valoraciones. Como primer paso, dividamos los datos de valoraciones en conjuntos de entrenamiento, validación y prueba:

```
import random
random.seed(0)
random.shuffle(ratings)
split1 = int(len(ratings) * 0.7)
split2 = int(len(ratings) * 0.85)
train = ratings[:split1]                # 70 % de los datos
validation = ratings[split1:split2]     # 15 % de los datos
test = ratings[split2:]                 # 15 % de los datos
```

Siempre es bueno tener un modelo de base sencillo y asegurarse de que el nuestro lo hace mejor. En este caso, un modelo básico podría ser “predecir la valoración media”. Utilizaremos el error medio cuadrado como medición, de forma que podamos comprobar qué tal se comporta el modelo básico con nuestro conjunto de datos:

```
avg_rating = sum(rating.rating for rating in train) / len(train)
baseline_error = sum((rating.rating–avg_rating) ** 2
                for rating in test) / len(test)
# Esto es lo que esperamos que haga mejor
assert 1.26 < baseline_error < 1.27
```

Teniendo nuestras incrustaciones, las valoraciones predichas vienen dadas por la matriz producto de las incrustaciones de usuario y las de película. Para un determinado usuario y película, ese valor no es más que el producto punto de las correspondientes incrustaciones.

Así que empecemos creando las incrustaciones. Las representaremos como clases `dict`, donde las claves son identificadores y los valores son vectores, lo que nos permitirá recuperar fácilmente la incrustación de un determinado identificador:

```
from scratch.deep_learning import random_tensor
EMBEDDING_DIM = 2
# Halla ID únicos
user_ids = {rating.user_id for rating in ratings}
movie_ids = {rating.movie_id for rating in ratings}
# Crea luego un vector aleatorio por ID
user_vectors = {user_id: random_tensor(EMBEDDING_DIM)
             for user_id in user_ids}
movie_vectors = {movie_id: random_tensor(EMBEDDING_DIM)
             for movie_id in movie_ids}
```

En este punto deberíamos ser ya bastante expertos en escribir bucles de entrenamiento:

```
from typing import List
import tqdm
from scratch.linear_algebra import dot
def loop(dataset: List[Rating],
 learning_rate: float = None) -> None:
   with tqdm.tqdm(dataset) as t:
      loss = 0.0
      for i, rating in enumerate(t):
         movie_vector = movie_vectors[rating.movie_id]
         user_vector = user_vectors[rating.user_id]
         predicted = dot(user_vector, movie_vector)
         error = predicted–rating.rating
         loss += error ** 2
         if learning_rate is not None:
            # predicted = m_0 * u_0 + ... + m_k * u_k
            # Así cada u_j introduce salida con coeficiente m_j
            # y cada m_j introduce salida con coeficiente u_j
            user_gradient = [error * m_j for m_j in movie_vector]
            movie_gradient = [error * u_j for u_j in user_vector]
            # Da pasos de gradiente
            for j in range(EMBEDDING_DIM):
               user_vector[j] -= learning_rate * user_gradient[j]
               movie_vector[j] -= learning_rate * movie_gradient[j]
         t.set_description(f"avg loss: {loss / (i + 1)}")
```

Ahora ya podemos entrenar nuestro modelo (es decir, encontrar las incrustaciones óptimas). En mi caso funcionó mejor si disminuía la velocidad de aprendizaje un poco en cada epoch:

```
learning_rate = 0.05
for epoch in range(20):
   learning_rate *= 0.9
   print(epoch, learning_rate)
   loop(train, learning_rate=learning_rate)
   loop(validation)
loop(test)
```

Este modelo es bastante apto para sobreajustar el conjunto de entrenamiento. Obtuve los mejores resultados con `EMBEDDING_DIM=2`, que me

proporcionó una pérdida media en el conjunto de prueba de más o menos 0.89.

> **Nota:** Si quisiera incrustaciones de muchas dimensiones, podría probar la regularización tal y como la usamos en la sección del mismo nombre del capítulo 15. En particular, en cada actualización de gradiente se podrían encoger los pesos hacia 0. No pude conseguir mejores resultados de ese modo.

A continuación, inspeccionamos los vectores aprendidos. No hay ninguna razón para esperar que los dos componentes tengan un significado especial, de modo que utilizaremos análisis de componente principal:

```
from scratch.working_with_data import pca, transform

original_vectors = [vector for vector in movie_vectors.values()]
components = pca(original_vectors, 2)
```

Transformemos nuestros vectores para que representen los componentes principales y conecten los identificadores de película y las valoraciones medias:

```
ratings_by_movie = defaultdict(list)
for rating in ratings:
   ratings_by_movie[rating.movie_id].append(rating.rating)
vectors = [
   (movie_id,
   sum(ratings_by_movie[movie_id]) / len(ratings_by_movie[movie_id]),
   movies[movie_id],
   vector)
   for movie_id, vector in zip(movie_vectors.keys(),
 transform(original_vectors, components))
]
# Imprime los primeros y últimos 25 por primer componente principal
print(sorted(vectors, key=lambda v: v[-1][0])[:25])
print(sorted(vectors, key=lambda v: v[-1][0])[-25:])
```

Los primeros 25 tienen todos valoraciones altas, mientras que los últimos 25 están valorados casi todos muy bajo (o ni siquiera están valorados en los datos de entrenamiento), lo que sugiere que el primer componente principal

está capturando en su mayoría “¿cómo de buena es esta película?”.

Es difícil para mí dar mucho sentido al segundo componente; de hecho, las incrustaciones bidimensionales funcionaron solo un poco mejor que las unidimensionales, lo que sugiere que, sea cual sea el segundo componente capturado, es posiblemente muy sutil (era de suponer que uno de los conjuntos de datos más grandes de MovieLens incluiría cosas más interesantes).

Para saber más

- Surprise, en `http://surpriselib.com/`, es una librería de Python para “construir y analizar sistemas recomendadores” que parece ser bastante popular y estar razonablemente actualizada.
- The Netflix Prize, en `https://en.wikipedia.org/wiki/Netflix_Prize`, fue una competición bastante conocida destinada a la creación de un sistema mejor para recomendar películas a los usuarios de Netflix.

24 Bases de datos y SQL

La memoria es el mejor amigo y el peor enemigo del hombre.

—Gilbert Parker

Los datos que necesitamos suelen residir en bases de datos, sistemas diseñados para almacenar y consultar datos de forma eficaz. La mayoría de ellas son bases de datos relacionales, como PostgreSQL, MySQL y SQL Server, que almacenan datos en tablas y se suelen consultar utilizando SQL (*Structured Query Language*, lenguaje de consulta estructurado), un lenguaje destinado específicamente a manipular datos.

El lenguaje SQL es una parte bastante esencial del kit de herramientas del científico de datos. En este capítulo crearemos NotQuiteABase, una implementación de Python de una estructura que no es del todo una base de datos. También abordaremos los fundamentos de SQL, mostrando al mismo tiempo cómo funciona en nuestra no-base de datos, que es la forma más "desde cero" que se me ha ocurrido para facilitar la comprensión de lo que estamos haciendo. Tengo la esperanza de que resolver problemas en NotQuiteABase sirva como una buena pista para averiguar cómo se podrían resolver los mismos problemas utilizando SQL.

CREATE TABLE e INSERT

Una base de datos relacional es una colección de tablas y de relaciones entre ellas. Una tabla es simplemente una colección de filas, no muy diferente a algunas de las matrices con las que hemos estado trabajando. Sin embargo, una tabla tiene también asociado un esquema fijo, que consiste en nombres de columna y tipos de columna.

Por ejemplo, supongamos un conjunto de datos `users` que contiene, para cada usuario, su `user_id`, `name` y `num_friends`:

```
users = [[0, "Hero", 0],
 [1, "Dunn", 2],
 [2, "Sue", 3],
 [3, "Chi", 3]]
```

En SQL podríamos crear esta tabla con:

```
CREATE TABLE users (
   user_id INT NOT NULL,
   name VARCHAR(200),
   num_friends INT);
```

Se puede observar que especificamos que `user_id` y `num_friends` deben ser números enteros (y que `user_id` no tiene permitido ser `NULL`, que indica un valor ausente y es parecido a nuestro `None`) y que el nombre debería ser una cadena de texto de longitud igual o menor a 200. Utilizaremos los tipos de Python del mismo modo.

> **Nota:** SQL es casi completamente insensible a las mayúsculas o minúsculas y a las sangrías de texto. En este libro ambas cosas son mi estilo preferido. Si empieza a estudiar SQL, seguro que encontrará otros ejemplos con estilos diferentes.

Podemos insertar las filas con sentencias `INSERT`:

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

Hay que tener en cuenta también que las sentencias SQL tienen que terminar en punto y coma, y que SQL requiere comillas simples para las cadenas de texto.

En NotQuiteABase crearemos una `Table` especificando un esquema parecido. Después, para insertar una fila, utilizaremos el método `insert` de la tabla, que toma una `list` de valores de fila que tienen que estar en el mismo orden que los nombres de columna de la tabla.

En segundo plano almacenaremos cada fila como una clase `dict` desde nombres de columna hasta valores. Una base de datos de verdad nunca utilizaría una representación que ocupara tanto espacio, pero hacerlo así nos

permitirá trabajar mucho mejor con NotQuiteABase.

Implementaremos la `Table` NotQuiteABase como una clase gigante, empleando un método cada vez. Empecemos quitándonos de en medio algunas importaciones y alias de tipo:

```
from typing import Tuple, Sequence, List, Any, Callable, Dict, Iterator
from collections import defaultdict
# Unos pocos alias de tipo que usaremos luego
Row = Dict[str, Any]                            # Una fila de base de datos
WhereClause = Callable[[Row], bool]             # Declara para una sola fila
HavingClause = Callable[[List[Row]], bool]      # Declara sobre varias filas
```

Empecemos por el constructor. Para crear una tabla NotQuiteABase, tendremos que pasarle una lista de nombres de columna y otra de tipos de columna, igual que haríamos si estuviéramos creando una tabla en una base de datos SQL:

```
class Table:
   def __init__(self, columns: List[str], types: List[type]) -> None:
      assert len(columns) == len(types), "# of            # of types"
      columns must ==
      self.columns = columns                                # Nombres de columnas
      self.types = types                                    # Tipos de datos de
                                                            columnas
      self.rows: List[Row] = [] # (sin datos aún)
```

Añadiremos un método auxiliar para obtener el tipo de una columna:

```
def col2type(self, col: str) -> type:
   idx = self.columns.index(col)      # Halla el índice de la columna,
   return self.types[idx]             # y devuelve su tipo.
```

Y añadiremos un método `insert` que compruebe que los valores que estamos insertando son válidos. En particular, tenemos que proporcionar el número correcto de valores, y cada uno tiene que ser el tipo correcto (o `None`):

```
def insert(self, values: list) -> None:
   # Comprueba que el nº de valores es correcto
```

```
    if len(values) != len(self.types):
        raise ValueError(f"You need to provide {len(self.types)} values")
    # Comprueba que los tipos de valores son correctos
    for value, typ3 in zip(values, self.types):
        if not isinstance(value, typ3) and value is not None:
            raise TypeError(f"Expected type {typ3} but got {value}")
    # Añade la correspondiente dict como una "fila"
    self.rows.append(dict(zip(self.columns, values)))
```

En una base de datos real se especificaría explícitamente si cualquier columna dada tenía permitido contener valores nulos (`None`); para simplificarnos las cosas simplemente diremos que cualquier columna puede.

Introduciremos además algunos métodos dunder que nos permitan tratar una tabla como una `List[Row]`, que utilizaremos principalmente para probar nuestro código:

```
def __getitem__(self, idx: int) -> Row:
    return self.rows[idx]
def __iter__(self) -> Iterator[Row]:
    return iter(self.rows)
def __len__(self) -> int:
    return len(self.rows)
```

Y añadiremos un método para mostrar en pantalla nuestra tabla:

```
def __repr__(self):
    """Pretty representation of the table: columns then rows"""
    rows = "\n".join(str(row) for row in self.rows)
    return f"{self.columns}\n{rows}"
```

Ahora podemos crear nuestra tabla `Users`:

```
# El constructor requiere nombres y tipos de columna
users = Table(['user_id', 'name', 'num_friends'], [int, str, int])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
```

```
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

Si ahora hacemos un `print(users)`, obtendremos:

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
```

La API de estilo lista permite escribir pruebas fácilmente:

```
assert len(users) == 11
assert users[1]['name'] == 'Dunn'
```

Tenemos mucha más funcionalidad que añadir.

UPDATE

Algunas veces necesitamos actualizar los datos que ya están en la base de datos. Por ejemplo, si Dunn adquiere otro amigo, quizá convendría hacer lo siguiente:

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

Las características más importantes son:

- Qué tabla actualizar.
- Qué filas actualizar.
- Qué campos actualizar.
- Cuáles deberían ser sus nuevos valores.

Añadiremos un método `update` similar a NotQuiteABase. Su primer argumento será una `dict`, cuyas claves son las columnas que se deben actualizar y cuyos valores son los nuevos valores para esos campos. Su segundo argumento (opcional) debería ser un `predicate` que devuelva `True` para las filas que deben ser actualizadas y `False` en otro caso.

```
def update(self,
           updates: Dict[str, Any],
           predicate: WhereClause = lambda row: True):
  # Comprueba que las actualizaciones tienen nombres y tipos válidos
  for column, new_value in updates.items():
    if column not in self.columns:
      raise ValueError(f"invalid column: {column}")
    typ3 = self.col2type(column)
    if not isinstance(new_value, typ3) and new_value is not None:
      raise TypeError(f"expected type {typ3}, but got {new_value}")
    # Y ahora actualiza
    for row in self.rows:
      if predicate(row):
        for column, new_value in updates.items():
          row[column] = new_value
```

Tras de lo cual podemos simplemente hacer esto:

```
assert users[1]['num_friends'] == 2            # Valor original
users.update({'num_friends' : 3},              # Fija num_friends = 3
          lambda row: row['user_id'] == 1)     # en filas donde user_id == 1
assert users[1]['num_friends'] == 3            # Valor actualizado
```

DELETE

Hay dos formas de borrar filas de una tabla en SQL. La que es peligrosa borra todas las filas de una tabla:

```
DELETE FROM users;
```

La manera menos peligrosa añade una cláusula `WHERE` y borra solamente las filas que cumplen una determinada condición:

```
DELETE FROM users WHERE user_id = 1;
```

Es fácil añadir esta funcionalidad a nuestra `Table`:

```
def delete(self, predicate: WhereClause = lambda row: True) -> None:
  """Delete all rows matching predicate"""
  self.rows = [row for row in self.rows if not predicate(row)]
```

Si proporcionamos una función `predicate` (por ejemplo, una cláusula `WHERE`), se borran únicamente las filas que la cumplen. Si no la incluimos, la `predicate` por defecto siempre devuelve `True`, y se borrarán todas las filas.

Por ejemplo:

```
# Realmente no ejecutaremos esto
users.delete(lambda row: row["user_id"] ==
1)                                          # Borra filas con user_id ==
users.delete()                              1
                                            # Borra todas las filas
```

SELECT

Normalmente las tablas SQL no se inspeccionan directamente. Lo que se suele hacer es consultarlas con una sentencia `SELECT`:

```
SELECT * FROM users;                        — toma el contenido completo
SELECT * FROM users LIMIT 2;                — toma las primeras dos
                                            filas
SELECT user_id FROM users;                  — solo toma ciertas columnas
SELECT user_id FROM users WHERE name =      — solo toma ciertas filas
‘Dunn’;
```

También se pueden utilizar sentencias `SELECT` para calcular campos:

```
SELECT LENGTH(name) AS name_length FROM users;
```

Le daremos a nuestra clase `Table` un método `select` que devuelva una nueva `Table`. El método acepta dos argumentos opcionales:

- `keep_columns` especifica los nombres de las columnas que se quieren mantener en el resultado. Si no se incluyen, el resultado contiene todas las columnas.
- `additional_columns` es un diccionario cuyas claves son nombres de nuevas columnas y cuyos valores son funciones que especifican cómo calcular los valores de las nuevas columnas. Echaremos un vistazo a las anotaciones de tipo de esas funciones para averiguar los tipos de las

nuevas columnas, de modo que las funciones tendrán que tener tipos de retorno anotados.

Si no incluyéramos ninguno de ellos, sencillamente obtendríamos de vuelta una copia de la tabla:

```
def select(self,
        keep_columns: List[str] = None,
        additional_columns: Dict[str, Callable] = None) -> 'Table':
   if keep_columns is None:          # Si no se especifican columnas,
      keep_columns =                 # devuelve todas
      self.columns
   if additional_columns is None:
      additional_columns = {}
   # Nombres y tipos de las nuevas columnas
   new_columns = keep_columns + list(additional_columns.keys())
   keep_types = [self.col2type(col) for col in keep_columns]
   # Así obtenemos el tipo de retorno de una anotación de tipo.
   # Fallará si 'calculation' no tiene un tipo de retorno.
   add_types = [calculation.__annotations__['return']
                              for calculation in
                              additional_columns.values()]
   # Crea una nueva tabla para los resultados
   new_table = Table(new_columns, keep_types + add_types)
   for row in self.rows:
      new_row = [row[column] for column in keep_columns]
      for column_name, calculation in additional_columns.items():
         new_row.append(calculation(row))
      new_table.insert(new_row)
   return new_table
```

Nota: ¿Recuerda, en el capítulo 2, cuando dijimos que las anotaciones de tipo en realidad no hacen nada? Pues bien, este es el ejemplo que lo desmiente. Pero eche un vistazo al enrevesado procedimiento que tenemos que pasar para llegar hasta ellas.

Nuestro `select` devuelve una nueva `Table`, mientras que el típico `SELECT` de SQL solo produce algo parecido a una serie de resultados temporales (a menos que se inserten explícitamente los resultados en una tabla).

También necesitaremos métodos `where` y `limit`, que son bastante

sencillos:

```
def where(self, predicate: WhereClause = lambda row: True) -> ‘Table’:
   """Return only the rows that satisfy the supplied predicate"""
   where_table = Table(self.columns, self.types)
   for row in self.rows:
      if predicate(row):
         values = [row[column] for column in self.columns]
         where_table.insert(values)
   return where_table
def limit(self, num_rows: int) -> ‘Table’:
   """Return only the first 'num_rows' rows"""
   limit_table = Table(self.columns, self.types)
   for i, row in enumerate(self.rows):
      if i >= num_rows:
         break
      values = [row[column] for column in self.columns]
      limit_table.insert(values)
   return limit_table
```

Después podemos construir fácilmente en NotQuiteABase equivalentes a las anteriores sentencias SQL:

```
# SELECT * FROM users;
all_users = users.select()
assert len(all_users) == 11
# SELECT * FROM users LIMIT 2;
two_users = users.limit(2)
assert len(two_users) == 2
# SELECT user_id FROM users;
just_ids = users.select(keep_columns=["user_id"])
assert just_ids.columns == [‘user_id’]
# SELECT user_id FROM users WHERE name = ‘Dunn’;
dunn_ids = (
   users
 .where(lambda row: row["name"] == "Dunn")
 .select(keep_columns=["user_id"])
)
assert len(dunn_ids) == 1
assert dunn_ids[0] == {"user_id": 1}
# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row) -> int: return len(row["name"])
name_lengths = users.select(keep_columns=[],
additional_columns = {"name_length": name_length})
```

```
assert name_lengths[0][‘name_length’] == len("Hero")
```

Tengamos en cuenta que para las consultas “fluidas” de varias líneas tenemos que poner entre paréntesis la consulta entera.

GROUP BY

Otra operación habitual en SQL es `GROUP BY`, que agrupa filas con valores idénticos en columnas determinadas y produce valores agregados como `MIN`, `MAX`, `COUNT` y `SUM`.

Por ejemplo, podría ocurrir que quisiéramos averiguar el número de usuarios y el `user_id` más pequeño para cada posible longitud de nombre:

```
SELECT LENGTH(name) as name_length,
   MIN(user_id) AS min_user_id,
   COUNT(*) AS num_users
FROM users
GROUP BY LENGTH(name);
```

Cada campo consultado con `SELECT` tiene que estar o bien en la cláusula `GROUP BY` (`name_length` lo está) o en un cálculo agregado (`min_user_id` y `num_users` lo están).

SQL soporta también una cláusula `HAVING`, que se comporta de forma similar a `WHERE`, excepto que su filtro se aplica a los agregados (mientras que `WHERE` filtraría filas antes siquiera de que la agregación tuviera lugar).

Quizá nos interese conocer el número medio de amigos para los usuarios cuyos nombres empiezan por determinadas letras, pero ver solo los resultados para letras cuya media correspondiente es mayor que 1 (sí, algunos de estos ejemplos son un poco forzados).

```
SELECT SUBSTR(name, 1, 1) AS first_letter,
   AVG(num_friends) AS avg_num_friends
FROM users
GROUP BY SUBSTR(name, 1, 1)
HAVING AVG(num_friends) > 1;
```

Nota: Las funciones para trabajar con cadenas de texto varían en las distintas implementaciones de SQL; puede que algunas bases de datos utilicen en su lugar `SUBSTRING` u otra distinta.

También se pueden calcular agregados totales. En ese caso, dejaríamos fuera `GROUP BY`:

```
SELECT SUM(user_id) as user_id_sum
FROM users
WHERE user_id > 1;
```

Para añadir esta funcionalidad a las `Table` de NotQuiteABase, incorporaremos un método `group_by`. Toma los nombres de las columnas por las que se desea agrupar, un diccionario de las funciones de agregación que se quieran ejecutar sobre cada grupo y un predicado opcional llamado `having` que opera sobre varias filas.

Después da los siguientes pasos:

1. Crea una subclase `defaultdict` para asignar colecciones `tuple` (de los valores por los que se quiere agrupar) a filas (que contienen los valores por los que se quiere agrupar). Conviene recordar que no se pueden utilizar listas como claves `dict`; hay que usar tuplas.
2. Itera por las filas de la tabla, poblando `defaultdict`.
3. Crea una nueva tabla con las columnas de salida correctas.
4. Itera por la `defaultdict` y llena la tabla de salida, aplicando el filtro `having`, si corresponde.

```
    def group_by(self,
group_by_columns: List[str],
aggregates: Dict[str, Callable],
having: HavingClause = lambda group: True) -> 'Table':
       grouped_rows = defaultdict(list)
       # Llena grupos
       for row in self.rows:
          key = tuple(row[column] for column in group_by_columns)
          grouped_rows[key].append(row)
       # La tabla final consiste en columnas group_by y agregados
       new_columns = group_by_columns + list(aggregates.keys())
```

```
        group_by_types = [self.col2type(col) for col in group_by_columns]
        aggregate_types = [agg.__annotations__['return']
for agg in aggregates.values()]
        result_table = Table(new_columns, group_by_types + aggregate_types)
        for key, rows in grouped_rows.items():
           if having(rows):
              new_row = list(key)
              for aggregate_name, aggregate_fn in aggregates.items():
                 new_row.append(aggregate_fn(rows))
              result_table.insert(new_row)
        return result_table
```

Nota: Una base de datos de verdad haría esto casi con total seguridad de un modo más eficiente.

De nuevo, veamos cómo haríamos el equivalente de las anteriores sentencias SQL. Las métricas `name_length` son:

```
def min_user_id(rows) -> int:
   return min(row["user_id"] for row in rows)
def length(rows) -> int:
   return len(rows)
stats_by_length = (
   users
   .select(additional_columns={"name_length" : name_length})
   .group_by(group_by_columns=["name_length"],
           aggregates={"min_user_id" : min_user_id,
                   "num_users" : length})
)
```

Las métricas `first_letter` son:

```
def first_letter_of_name(row: Row) -> str:
   return row["name"][0] if row["name"] else ""
def average_num_friends(rows: List[Row]) -> float:
   return sum(row["num_friends"] for row in rows) / len(rows)
def enough_friends(rows: List[Row]) -> bool:
   return average_num_friends(rows) > 1
avg_friends_by_letter = (
   users
   .select(additional_columns={'first_letter' : first_letter_of_name})
   .group_by(group_by_columns=['first_letter'],
```

```
 aggregates={"avg_num_friends" : average_num_friends},
 having=enough_friends)
)
```

Y `user_id_sum` es:

```
def sum_user_ids(rows: List[Row]) -> int:
   return sum(row["user_id"] for row in rows)
user_id_sum = (
   users
   .where(lambda row: row["user_id"] > 1)
   .group_by(group_by_columns=[],
 aggregates={ "user_id_sum" : sum_user_ids })
)
```

ORDER BY

Con frecuencia nos vendrá bien ordenar los resultados. Por ejemplo, quizá nos interese conocer los primeros dos nombres (en orden alfabético) de nuestros usuarios:

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

Esto es fácil de implementar, dando a nuestra `Table` un método `order_by` que toma una función `order`:

```
def order_by(self, order: Callable[[Row], Any]) -> ‘Table’:
     new_table = self.select()          # hace una copia
     new_table.rows.sort(key=order)
     return new_table
```

Que podemos utilizar después del siguiente modo:

```
friendliest_letters = (
   avg_friends_by_letter
   .order_by(lambda row: -row["avg_num_friends"])
   .limit(4)
)
```

La cláusua `ORDER BY` de SQL permite especificar `ASC` (ascendente) o `DESC` (descendente) para cada campo de ordenación; aquí tendríamos que incluirlo en nuestra función `order`.

JOIN

Las tablas de bases de datos relacionales suelen estar normalizadas, lo que significa que están organizadas para minimizar la redundancia. Por ejemplo, cuando trabajamos con los intereses de nuestros usuarios en Python, podemos asignar a cada usuario una `list` que contenga sus intereses.

Las tablas en SQL no pueden contener listas, de modo que la solución habitual es crear una segunda tabla llamada `user_interests`, que contenga la relación de uno a muchos entre `user_id` e `interest`. En SQL haríamos esto:

```
CREATE TABLE user_interests (
   user_id INT NOT NULL,
   interest VARCHAR(100) NOT NULL
);
```

Mientras que en NotQuiteABase, crearíamos la tabla:

```
user_interests = Table([‘user_id’, ‘interest’], [int, str])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

Nota: Sigue habiendo mucha redundancia (el interés “SQL” está almacenado en dos lugares distintos). En una base de datos real se podría almacenar `user_id` e `interest_id` en la tabla `user_interests`, y después crear una tercera tabla, `interests`, que asigne `interest_id` a `interest`, de forma que solo se pudiera almacenar una muestra del nombre de cada interés en cada ocasión. En nuestro caso, lo único que esto conseguiría es que nuestros ejemplos fueran más complicados de lo necesario.

Cuando nuestros datos residen en distintas tablas, ¿cómo los analizamos?

Utilizando `JOIN` para unir las tablas. `JOIN` combina las filas de la tabla izquierda con las filas correspondientes de la tabla derecha, donde el significado de "correspondiente" se basa en cómo especificamos la unión.

Por ejemplo, para encontrar los usuarios interesados en SQL haríamos esta consulta:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

`JOIN` indica que, por cada fila de `users`, deberíamos mirar su `user_id` y asociar dicha fila a cada fila de `user_interests` que contenga el mismo `user_id`.

Conviene observar que tuvimos que especificar qué tablas unir con `JOIN` y en qué columnas con la cláusula `ON`. Esto se denomina `INNER JOIN`, que devuelve las combinaciones de filas (y solo las combinaciones de filas) que coinciden según el criterio de unión especificado.

También existe `LEFT JOIN`, que (además de las combinaciones de filas coincidentes) devuelve una fila por cada fila de la tabla izquierda sin filas coincidentes (en cuyo caso, los campos que habrían venido de la tabla derecha son todos `NULL`).

Utilizando `LEFT JOIN`, es fácil contar el número de intereses que tiene cada usuario:

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

`LEFT JOIN` asegura que los usuarios sin intereses seguirán teniendo filas en el conjunto de datos combinado (con valores `NULL` para los campos que vengan de `user_interests`), y `COUNT` cuenta solo valores que son no `NULL`.

La implementación de `join` en NotQuiteABase será más restrictiva (simplemente une dos tablas según cualesquiera columnas que tengan en común). Aún así, no es algo sencillo de escribir:

```
  def join(self, other_table: ‘Table’, left_join: bool = False) -> ‘Table’:
    join_on_columns = [c for c in                # columnas de
    self.columns
                if c in                          # ambas tablas
                other_table.columns]
    additional_columns = [c for c in             # columnas solo
    other_table.columns
                if c not in                      # de la tabla derecha
                join_on_columns]
    # todas las columnas de la tabla izqda + additional_columns de la derecha
    new_columns = self.columns + additional_columns
    new_types = self.types + [other_table.col2type(col)
                      for col in additional_columns]
    join_table = Table(new_columns, new_types)
    for row in self.rows:
      def is_join(other_row):
        return all(other_row[c] == row[c] for c in join_on_columns)
      other_rows = other_table.where(is_join).rows
      # Las otras filas que coinciden con esta producen una fila de
      resultado.
      for other_row in other_rows:
        join_table.insert([row[c] for c in self.columns] +
                      [other_row[c] for c in additional_columns])
      # Si no hay ninguna fila y es un left join, el resultado es None.
      if left_join and not other_rows:
        join_table.insert([row[c] for c in self.columns] +
                      [None for c in additional_columns])
    return join_table
```

Así, podríamos encontrar usuarios interesados en SQL con:

```
sql_users = (
  users
  .join(user_interests)
  .where(lambda row: row["interest"] == "SQL")
  .select(keep_columns=["name"])
)
```

Y podríamos obtener los recuentos de intereses con:

```
def count_interests(rows: List[Row]) -> int:
  """counts how many rows have non-None interests"""
  return len([row for row in rows if row["interest"] is not None])
user_interest_counts = (
  users
```

```
   .join(user_interests, left_join=True)
   .group_by(group_by_columns=["user_id"],
        aggregates={"num_interests" : count_interests })
)
```

En SQL, hay también una cláusula `RIGHT JOIN`, que mantiene filas de la tabla derecha que no tienen ninguna coincidencia, y una `FULL OUTER JOIN`, que mantiene filas de ambas tablas que tampoco tienen coincidencias. No implementaremos ninguna de ellas.

Subconsultas

En SQL se puede utilizar `SELECT` para elegir (y combinar con `JOIN`) de entre los resultados de consultas como si fueran tablas. Así, si quisiéramos encontrar el `user_id` más pequeño de cualquiera interesado en SQL, podríamos utilizar una subconsulta (por supuesto, se podría hacer el mismo cálculo empleando `JOIN`, pero así no sería posible ilustrar las subconsultas).

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = ‘SQL’) sql_interests;
```

Dado el modo en que hemos diseñado NotQuiteABase, esto nos sale gratis (los resultados de nuestra consulta son tablas reales).

```
likes_sql_user_ids = (
   user_interests
   .where(lambda row: row["interest"] == "SQL")
   .select(keep_columns=[‘user_id’])
)
likes_sql_user_ids.group_by(group_by_columns=[],
aggregates={ "min_user_id" : min_user_id })
```

Índices

Para encontrar filas que contengan un determinado valor (digamos, en las

que `name` es “Hero”), NotQuiteABase tiene que inspeccionar cada fila de la tabla. Si la tabla tiene muchas filas, ello puede requerir mucho tiempo.

De forma similar, nuestro algoritmo `join` es extremadamente ineficaz. Por cada fila de la tabla izquierda, inspecciona cada fila de la tabla derecha para ver si coinciden. Con dos tablas grandes, este proceso podría durar aproximadamente para siempre.

Además, muchas veces querríamos aplicar restricciones a algunas de las columnas. Por ejemplo, en la tabla `users` probablemente no nos interesa permitir que dos usuarios distintos tengan el mismo `user_id`.

Los índices resuelven estos problemas. Si la tabla `user_interests` tuviera un índice sobre `user_id`, un algoritmo `join` inteligente podría hallar coincidencias directamente en lugar de explorando toda la tabla. Si la tabla `users` tuviera un índice “único” sobre `user_id`, se obtendría un error al intentar insertar un duplicado.

Cada tabla de una base de datos puede tener uno o varios índices, que permiten consultar rápidamente filas por columnas clave, combinar tablas de manera eficaz y aplicar restricciones únicas a columnas o a combinaciones de columnas.

Diseñar y utilizar índices bien es algo parecido a la magia negra (que varía dependiendo de la base de datos que se tenga entre manos), pero, si finalmente se consigue trabajar mucho con base de datos, vale la pena estudiarlos.

Optimización de consultas

Recordemos la consulta para hallar todos los usuarios que están interesados en SQL:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = ‘SQL’
```

En NotQuiteABase hay (al menos) dos formas diferentes de escribir esta

consulta. Se podría filtrar la tabla `user_interests` realizando la unión:

```
(
  user_interests
  .where(lambda row: row["interest"] == "SQL")
  .join(users)
  .select(["name"])
)
```

O también se podrían filtrar los resultados de la combinación:

```
(
  user_interests
  .join(users)
  .where(lambda row: row["interest"] == "SQL")
  .select(["name"])
)
```

Terminaremos con los mismos resultados con cualquiera de los métodos, pero el último de filtrar antes de combinar es casi seguro más eficaz, ya que en ese caso `join` tiene muchas menos filas con la que trabajar.

En SQL en general no hace falta preocuparse de esto. Se “declaran” los resultados deseados y se deja al motor de consulta ejecutarlos (y utilizar índices de forma eficiente).

NoSQL

Una reciente tendencia en las bases de datos es la de las bases de datos “NoSQL” no relacionales, que no representan los datos en tablas. Por ejemplo, MongoDB es una conocida base de datos sin esquemas cuyos elementos son documentos JSON arbitrariamente complejos en lugar de filas.

Hay bases de datos de columnas que almacenan datos en columnas en lugar de filas (algo bueno cuando los datos tienen muchas columnas, pero las consultas necesitan pocas), almacenes de clave/valor que se optimizan para recuperar (complejos) valores únicos por sus claves, bases de datos para almacenar y atravesar grafos, bases de datos optimizadas para funcionar a lo

largo de varios centros de datos, bases de datos diseñadas para ejecutarse en memoria, bases de datos para almacenar datos de series temporales, y muchas más.

La tendencia del mañana puede que ni siquiera exista hoy, de modo que no puedo hacer mucho más que hacerle saber que NoSQL existe. Así que ya lo sabe. Es algo que existe.

Para saber más

- Si quiere descargar una base de datos para jugar, SQLite, en `http://www.sqlite.org`, es rápida y de tamaño reducido, mientras que MySQL, en `http://www.mysql.com`, y PostgreSQL, en `http://www.postgresql.org`, son más grandes y están repletas de funciones. Las tres son gratuitas y tienen mucha documentación.
- Si quiere explorar NoSQL, MongoDB, en `http://www.mongodb.org`, es muy sencilla para empezar, cosa que puede ser en parte una bendición y en parte una maldición. También tiene bastante buena documentación.
- El artículo de Wikipedia sobre NoSQL, en `https://es.wikipedia.org/wiki/NoSQL`, contiene ahora casi seguro enlaces a bases de datos que ni siquiera existían cuando se escribió este libro.

25 MapReduce

El futuro ya ha llegado. Es solo que aún no está equitativamente repartido.

—William Gibson

MapReduce es un modelo de programación para realizar procesos paralelos con grandes conjuntos de datos. Aunque es una técnica muy potente, sus fundamentos son relativamente sencillos.

Supongamos que tenemos una colección de elementos que queremos procesar de alguna forma. Por ejemplo, los elementos podrían ser registros de sitios web, textos de varios libros, archivos de imágenes o cualquier otra cosa. Una versión básica del algoritmo MapReduce consiste en los siguientes pasos:

- Utilizar una función `mapper` para convertir cada elemento en cero o en más pares clave/valor (a menudo a esto se le llama función `map`, pero ya hay una función de Python denominada `map` y no queremos confundirlas).
- Recopilar todos los pares con claves idénticas.
- Aplicar una función `reducer` a cada colección de valores agrupados para producir valores de salida para la clave correspondiente.

Nota: MapReduce está tan pasado de moda, que pensé en quitar este capítulo entero de la segunda edición. Pero decidí que seguía siendo un tema interesante, así que finalmente acabé dejándolo (obviamente).

Todo esto suena muy abstracto, así que veamos un ejemplo específico. Hay pocas reglas absolutas en ciencia de datos, pero una de ellas es que nuestro primer ejemplo de MapReduce tiene que ver con contar palabras.

Ejemplo: Recuento de palabras

¡DataSciencester ha crecido, y ahora tiene millones de usuarios! Esto es estupendo para su seguridad laboral, pero dificulta ligeramente los análisis rutinarios.

Por ejemplo, su vicepresidente de Contenido quiere saber de qué cosas habla la gente en sus actualizaciones de estado. Como primer intento, decide contar las palabras que aparecen, de modo que pueda preparar un informe sobre las más frecuentes.

Cuando había varios cientos de usuarios, esto era fácil de hacer:

```
from typing import List
from collections import Counter
def tokenize(document: str) -> List[str]:
   """Just split on whitespace"""
   return document.split()
def word_count_old(documents: List[str]):
   """Word count not using MapReduce"""
   return Counter(word
      for document in documents
      for word in tokenize(document))
```

Con millones de usuarios, el conjunto de `documents` (actualizaciones de estado) resulta de repente demasiado grande para manejarlo en el ordenador. Pero, si se puede ajustar al modelo MapReduce, podría utilizar una cierta infraestructura de Big Data que sus ingenieros han implementado.

Lo primero que necesitamos es una función que convierta un documento en una secuencia de pares de clave/valor. Queremos que nuestros resultados se agrupen por palabra, lo que significa que las claves deberían ser palabras. Para cada palabra, además, emitiremos el valor 1 para indicar que este par corresponde a una aparición de la palabra:

```
from typing import Iterator, Tuple
def wc_mapper(document: str) -> Iterator[Tuple[str, int]]:
   """For each word in the document, emit (word, 1)"""
   for word in tokenize(document):
      yield (word, 1)
```

Saltándonos por el momento el repetitivo paso 2, supongamos que para una palabra hemos recogido una lista de los correspondientes recuentos que emitimos. Entonces, para producir el recuento total para esa palabra, tan solo necesitamos:

```
from typing import Iterable
def wc_reducer(word: str,
          counts: Iterable[int]) -> Iterator[Tuple[str, int]]:
   """Sum up the counts for a word"""
   yield (word, sum(counts))
```

Volviendo al paso 2, ahora tenemos que recopilar los resultados de `wc_mapper` y pasárselos a `wc_reducer`. Pensemos en cómo haríamos esto en un solo ordenador:

```
from collections import defaultdict
def word_count(documents: List[str]) -> List[Tuple[str, int]]:
   """Count the words in the input documents using MapReduce"""
   collector = defaultdict(list)      # To store grouped values
   for document in documents:
      for word, count in wc_mapper(document):
         collector[word].append(count)
   return [output
         for word, counts in collector.items()
         for output in wc_reducer(word, counts)]
```

Supongamos que tenemos tres documentos `[“data science”, “big data”, “science fiction”]`.

Así, `wc_mapper` aplicado al primer documento produce los dos pares `(“data”, 1)` y `(“science”, 1)`. Una vez hemos pasado por los tres documentos, el `collector` contiene:

```
{"data" : [1, 1],
 "science" : [1, 1],
 "big" : [1],
 "fiction" : [1]}
```

Entonces `wc_reducer` produce los recuentos de cada palabra:

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

¿Por qué MapReduce?

Como ya dijimos anteriormente, el principal beneficio de MapReduce es que nos permite distribuir cálculos moviendo el procesamiento a los datos. Supongamos que queremos contar las palabras en miles de millones de documentos.

Nuestro planteamiento original (no con MapReduce) requiere que el ordenador sea el que se encargue de procesar el acceso a cada documento. Esto significa que los documentos tienen todos que estar en ese equipo o bien ser transferidos a él durante el proceso. También significa, lo que es más importante, que el ordenador solo puede procesar un documento cada vez.

Nota: Posiblemente pueda procesar unos cuantos cada vez si tiene varios procesadores y si el código se ha reescrito para aprovecharlos. Pero, aun así, todos los documentos tendrán que acabar estando en ese equipo.

Imaginemos ahora que nuestros miles de millones de documentos están repartidos en 100 ordenadores. Con la infraestructura adecuada (y pasando por alto algunos de los detalles), podemos hacer lo siguiente:

- Que cada equipo ejecute el mapeador en sus documentos, produciendo así muchos pares clave/valor.
- Distribuir esos pares clave/valor en una serie de ordenadores “reductores”, asegurándose de que los pares correspondientes a cualquier clave determinada terminan todos en el mismo equipo.
- Que cada ordenador reductor agrupe los pares por clave y ejecute a continuación el reductor en cada conjunto de valores.
- Devolver cada par (clave, salida).

Lo que resulta sorprendente de esto es que dimensiona horizontalmente. Si duplicamos el número de equipos, entonces (ignorando determinados costes fijos de ejecutar un sistema MapReduce) nuestro cálculo debería realizarse aproximadamente el doble de rápido. Cada equipo mapeador solo necesitará hacer la mitad del trabajo, y (suponiendo que haya suficientes claves distintas

para distribuir aún más el trabajo del reductor) lo mismo ocurre con los ordenadores reductores.

MapReduce, más general

Si pensamos en ello un momento, todo el código específico para contar palabras del ejemplo anterior está contenido en las funciones `wc_mapper` y `wc_reducer`. Esto significa que con un par de cambios tenemos una estructura mucho más general (que sigue funcionando en un solo equipo).

Podríamos utilizar tipos genéricos para anotar completamente los tipos de nuestra función `map_reduce`, pero terminaría siendo un lío pedagógicamente hablando, de modo que en este capítulo nos despreocuparemos mucho más de nuestras anotaciones de tipo:

```
from typing import Callable, Iterable, Any, Tuple

# Un par clave/valor es solo una tupla de dos
KV = Tuple[Any, Any]

# Mapper es una función que devuelve un Iterable de pares clave/valor
Mapper = Callable[..., Iterable[KV]]

# Reducer es una función que toma una clave y un iterable de valores
# y devuelve un par clave/valor
Reducer = Callable[[Any, Iterable], KV]
```

Ahora podemos escribir una función general `map_reduce`:

```
def map_reduce(inputs: Iterable,
               mapper: Mapper,
               reducer: Reducer) -> List[KV]:
    """Run MapReduce on the inputs using mapper and reducer"""
    collector = defaultdict(list)
    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)
    return [output
            for key, values in collector.items()
            for output in reducer(key, values)]
```

Después podemos contar palabras simplemente utilizando:

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

Lo que nos da flexibilidad para resolver una amplia variedad de problemas.

Antes de continuar, tengamos en cuenta que `wc_reducer` está simplemente sumando los valores correspondientes a cada clave. Este tipo de agregación es tan habitual, que vale la pena generalizarlo:

```
def values_reducer(values_fn: Callable) -> Reducer:
   """Return a reducer that just applies values_fn to its values"""
   def reduce(key, values: Iterable) -> KV:
      return (key, values_fn(values))
   return reduce
```

Tras de lo cual podemos crear fácilmente:

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))

assert sum_reducer("key", [1, 2, 3, 3]) == ("key", 9)
assert min_reducer("key", [1, 2, 3, 3]) == ("key", 1)
assert max_reducer("key", [1, 2, 3, 3]) == ("key", 3)
assert count_distinct_reducer("key", [1, 2, 3, 3]) == ("key", 3)
```

Y así sucesivamente.

Ejemplo: Analizar actualizaciones de estado

El vicepresidente de Contenido ha quedado impresionado por los recuentos de palabras y le pregunta qué más puede averiguar de las actualizaciones de estado de la gente, así que se las arregla para extraer un conjunto de datos de actualizaciones de estado que tiene este aspecto:

```
status_updates = [
   {"id": 2,
   "username" : "joelgrus",
   "text" : "Should I write a second edition of my data science book?",
```

```
   "created_at" : datetime.datetime(2018, 2, 21, 11, 47, 0),
   "liked_by" : ["data_guy", "data_gal", "mike"] },
   # ...
]
```

Digamos que tenemos que averiguar qué día de la semana habla más la gente sobre ciencia de datos. Para descubrirlo, simplemente contaremos cuántas actualizaciones de ciencia de datos hay cada día de la semana. Esto significa que tendremos que agrupar por el día de la semana, así que esa es nuestra clave. Si emitimos un valor de 1 para cada actualización que contenga "*data science*", simplemente podemos obtener el número total utilizando `sum`:

```
def data_science_day_mapper(status_update: dict) -> Iterable:
   """Yields (day_of_week, 1) if status_update contains "data science" """
   if "data science" in status_update["text"].lower():
      day_of_week = status_update["created_at"].weekday()
      yield (day_of_week, 1)
data_science_days = map_reduce(status_updates,
 data_science_day_mapper,
 sum_reducer)
```

Como ejemplo un poco más complicado, imaginemos que queremos averiguar para cada usuario la palabra más común que pone en sus actualizaciones de estado. Hay tres posibles planteamientos que me vienen a la mente para el `mapper`:

- Poner el nombre de usuario en la clave; poner las palabras y los recuentos en los valores.
- Poner la palabra en la clave; poner los nombres de usuario y los recuentos en los valores.
- Poner el nombre de usuario y la palabra en la clave; poner los recuentos en los valores.

Si pensamos en ello un poco más, definitivamente nos interesa agrupar por `username`, porque queremos considerar las palabras de cada persona por separado. No nos viene bien agrupar por `word`, ya que nuestro reductor necesitará ver todas las palabras para cada persona para descubrir cuál es la

más popular, lo que significa que la primera opción es la correcta:

```
def words_per_user_mapper(status_update: dict):
   user = status_update["username"]
   for word in tokenize(status_update["text"]):
      yield (user, (word, 1))
def most_popular_word_reducer(user: str,
 words_and_counts: Iterable[KV]):
   """
   Given a sequence of (word, count) pairs,
   return the word with the highest total count
   """
   word_counts = Counter()
   for word, count in words_and_counts:
      word_counts[word] += count
   word, count = word_counts.most_common(1)[0]
   yield (user, (word, count))
user_words = map_reduce(status_updates,
                  words_per_user_mapper,
                  most_popular_word_reducer)
```

O bien podríamos averiguar el número de personas diferentes a las que le gusta el estado para cada usuario:

```
def liker_mapper(status_update: dict):
   user = status_update["username"]
   for liker in status_update["liked_by"]:
      yield (user, liker)
distinct_likers_per_user = map_reduce(status_updates,
 liker_mapper,
 count_distinct_reducer)
```

Ejemplo: Multiplicación de matrices

Recuerde de la subsección “Multiplicación de matrices” del capítulo 22 que, dada una matriz `A [n, m]` y otra `B [m, k]`, podemos multiplicarlas para formar una matriz `C [n, k]`, donde el elemento de `C` de la fila *i* y columna *j* viene dado por:

```
C[i][j] = sum(A[i][x] * B[x][j] for x in range(m))
```

Esto funciona si representamos nuestras matrices como listas de listas, como hemos estado haciendo.

Pero las matrices grandes son en ocasiones dispersas, lo que significa que la mayoría de sus elementos son iguales a 0. Para grandes matrices dispersas, una lista de listas puede ser una representación muy poco eficaz. Una representación más compacta solo almacena las ubicaciones con valores no cero:

```
from typing import NamedTuple
class Entry(NamedTuple):
   name: str
   i: int
   j: int
   value: float
```

Por ejemplo, una matriz de 1.000 x 1.000 millones tiene 1 quintillón de entradas, que no serían fáciles de almacenar en un ordenador. Pero, si solo hay algunas entradas no cero en cada fila, esta representación alternativa es muchas órdenes de magnitud más pequeña.

Dado este tipo de representación, resulta que podemos utilizar MapReduce para realizar multiplicación de matrices de una forma distribuida.

Para motivar a nuestro algoritmo, podemos observar que cada elemento `A[i][j]` solo se utiliza para calcular los elementos de `C` en la fila `i`, y cada elemento `B[i][j]` solo se utiliza para calcular los elementos de `C` en la columna `j`. Nuestro objetivo será que cada salida de nuestro `reducer` sea una sola entrada de `C`, lo que significa que necesitaremos nuestro mapeador para emitir claves que identifiquen una sola entrada de `C`. Esto sugiere lo siguiente:

```
def matrix_multiply_mapper(num_rows_a: int, num_cols_b: int) -> Mapper:
   # C[x][y] = A[x][0] * B[0][y] + ... + A[x][m] * B[m][y]
   #
   # así un elemento A[i][j] va dentro de cada C[i][y] con coef B[j][y]
   # y un elemento B[i][j] va dentro de cada C[x][j] con coef A[x][i]
   def mapper(entry: Entry):
      if entry.name == "A":
         for y in range(num_cols_b):
            key = (entry.i, y)                    # qué elemento de C
            value = (entry.j, entry.value)        # qué entrada de la suma
```

```
          yield (key, value)
        else:
          for x in range(num_rows_a):
            key = (x, entry.j)                  # qué elemento de C
            value = (entry.i, entry.value)      # qué entrada de la suma
            yield (key, value)
    return mapper
```

Y después:

```
def matrix_multiply_reducer(key: Tuple[int, int],
indexed_values: Iterable[Tuple[int, int]]):
  results_by_index = defaultdict(list)
  for index, value in indexed_values:
    results_by_index[index].append(value)
  # Multiplica los valores por posiciones con dos valores
  # (uno de A, y uno de B) y los suma.
  sumproduct = sum(values[0] * values[1]
               for values in results_by_index.values()
               if len(values) == 2)
  if sumproduct != 0.0:
    yield (key, sumproduct)
```

Por ejemplo, si tuviéramos estas dos matrices:

```
A = [[3, 2, 0],
 [0, 0, 0]]
B = [[4, -1, 0],
 [10, 0, 0],
 [0, 0, 0]]
```

Podríamos reescribirlas como tuplas:

```
entries = [Entry("A", 0, 0, 3), Entry("A", 0, 1, 2), Entry("B", 0, 0, 4),
        Entry("B", 0, 1, -1), Entry("B", 1, 0, 10)]
mapper = matrix_multiply_mapper(num_rows_a=2, num_cols_b=3)
reducer = matrix_multiply_reducer
# El producto debería ser [[32, -3, 0], [0, 0, 0]].
# Así debería tener dos entradas.
assert (set(map_reduce(entries, mapper, reducer)) ==
     {((0, 1), -3), ((0, 0), 32)})
```

Esto no es terriblemente interesante en unas matrices tan pequeñas, pero, si tuviéramos millones de filas y millones de columnas, podría ayudar mucho.

Un inciso: Combinadores

Una cosa que quizá haya notado es que muchos de nuestros mapeadores parecen incluir mucha información adicional. Por ejemplo, al contar palabras, en lugar de emitir `(word, 1)` y sumar los valores, podríamos haber emitido `(word, None)` y simplemente tomado la longitud. Una razón por la que no hicimos esto es que, en la configuración distribuida, en ocasiones queremos utilizar combinadores para reducir la cantidad de datos que se tienen que transferir de un equipo a otro. Si uno de nuestros equipos mapeadores ve la palabra datos 500 veces, podemos decirle que combine las 500 apariciones de `("datos", 1)` en un solo `("datos", 500)` antes de pasarlo al equipo reductor. Esto da como resultado poder mover muchos menos datos, lo que puede conseguir que nuestro algoritmo sea notablemente más rápido.

Debido al modo en que escribimos nuestro reductor, manejaría estos datos combinados correctamente (si lo hubiéramos escrito usando `len`, no lo habría hecho).

Para saber más

- Como dije al principio, MapReduce es ahora mucho menos popular que cuando escribí la primera edición. Probablemente no vale la pena invertir mucho tiempo en él.
- Dicho esto, el sistema de MapReduce más utilizado es Hadoop, en `http://hadoop.apache.org`. Existen varias distribuciones comerciales y no comerciales y un enorme ecosistema de herramientas relacionadas con Hadoop.
- Amazon.com ofrece un servicio Elastic MapReduce, en `https://aws.amazon.com/es/emr/`, que es probablemente más fácil

que configurar su propio clúster.

- Los trabajos de Hadoop suelen tener una elevada latencia, lo que les convierte en una mala opción para análisis en “tiempo real”. Una opción habitual para estas cargas de trabajo es Spark, en `http://spark.apache.org/`, que puede ser muy MapReduce.

26 La ética de los datos

Comida primero, ética después.

—Bertolt Brecht

¿Qué es la ética de los datos?

El uso de los datos trae aparejado el mal uso de los mismos, lo que ha ocurrido prácticamente siempre. Pero, últimamente, esta idea ha sido etiquetada como "ética de los datos", y aparece de forma bastante destacada en las noticias.

Por ejemplo, en las elecciones presidenciales de EE. UU. de 2016, una compañía llamada Cambridge Analytica accedió indebidamente a datos de Facebook,[1] y los utilizó para orientar la publicidad política.

En 2018, un coche autónomo que estaba siendo probado por Uber golpeó y mató a un peatón[2] (había un "conductor de seguridad" en el coche, pero parece ser que no estaba prestando atención en ese momento).

Se utilizan algoritmos para predecir el riesgo de que los criminales reincidan,[3] y para sentenciarles en consecuencia. ¿Es esto más o menos justo que permitir a los jueces determinar lo mismo?

Algunas líneas aéreas asignan a las familias asientos separados,[4] obligándoles a pagar más por sentarse juntos. ¿Ha intentado algún científico de datos evitar esto?

La "ética de los datos" pretende ofrecer respuestas a estas preguntas, o al menos una estrategia para lidiar con ellas. Yo no soy tan arrogante como para decirle cómo tiene que pensar sobre estos hechos (que cambian rápidamente), de modo que en este capítulo haremos un recorrido rápido por algunos de los temas más importantes y espero inspirar a mis lectores para que reflexionen sobre ellos (además, no soy tan buen filósofo como para hacer ética desde

cero).

No, ahora en serio, ¿qué es la ética de datos?

Pues bien, empecemos por tratar de responder a la pregunta "¿qué es la ética?". Si combinamos todas las definiciones que se pueden encontrar, acabaremos con algo como que la ética es un marco de reflexión sobre el comportamiento "correcto" e "incorrecto"; por tanto, la ética de datos es un marco de reflexión sobre el comportamiento correcto e incorrecto relacionado con los datos.

Algunas personas hablan como si la "ética de los datos" fuera (quizá de forma implícita) una serie de mandamientos sobre lo que se puede y no se puede hacer. Algunas de ellas trabajan duro creando manifiestos, otras elaborando compromisos obligatorios que esperan que los demás cumplan. Todavía hay otros que hacen campaña para que la ética de los datos sea una parte obligatoria del currículum en ciencia de datos (de ahí este capítulo, como medio de cubrir mis apuestas en caso de que lo logren).

> **Nota:** Curiosamente, no hay muchos datos que sugieran que los cursos de ética realmente conducen a un comportamiento ético (`https://www.washingtonpost.com/news/on-leadership/wp/2014/01/13/can-you-teach-businessmen-to-be-ethical`), en cuyo caso, ¡quizá esta campaña sea en sí misma poco ética!

Otras personas (por ejemplo, servidor) creen que la gente razonable normalmente no se pone de acuerdo sobre cuestiones sutiles acerca de lo correcto y lo incorrecto, y que lo realmente importante de la ética de los datos es comprometerse a tener en cuenta las consecuencias éticas de nuestros comportamientos. Esto obliga a entender el tipo de cosas que muchos defensores de la ética de datos no aprueban, pero no necesariamente requiere estar de acuerdo con su desaprobación.

¿Debo preocuparme de la ética de los datos?

Sí, debe preocuparse de la ética de los datos sea cual sea su trabajo. Si tiene que ver con datos, es libre de darle a su preocupación el carácter de "ética de los datos", pero debería preocuparse lo mismo por la ética de las partes de su trabajo que no tienen que ver con datos.

Quizá lo distinto en los trabajos que tienen que ver con la tecnología es que la tecnología dimensiona, y que las decisiones tomadas por individuos que trabajan en problemas tecnológicos (estén o no relacionados con datos) tienen efectos que pueden llegar muy lejos.

Un pequeño cambio en un algoritmo de descubrimiento de noticias podría suponer la diferencia entre que millones de personas lean un artículo o que nadie lo lea.

Un algoritmo defectuoso para conceder la libertad condicional que se utilice en todo el país afecta sistemáticamente a millones de personas, mientras que una junta de libertad condicional defectuosa a su manera solo afecta a las personas que se presentan ante ella.

Así que, sí, en general, debe preocuparse de los efectos que tiene su trabajo en el mundo. Y cuanto más lejos lleguen los efectos de su trabajo, más tendrá que preocuparse de estas cosas.

Lamentablemente, parte del discurso sobre la ética de los datos implica a personas que tratan de imponer a los demás sus conclusiones éticas. Pero el que usted se preocupe de las mismas cosas que a ellos les preocupan es cosa suya.

Crear productos de datos de mala calidad

Algunas cuestiones relativas a la "ética de los datos" son el resultado de crear productos de mala calidad.

Por ejemplo, Microsoft lanzó un bot de conversación llamado Tay,[5] que repetía como un papagayo todo lo que se le tuiteaba. Internet rápidamente descubrió que era posible hacer que Tay dijera todo tipo de mensajes

ofensivos. Parecía improbable que nadie en Microsoft discutiera la ética de lanzar un bot “racista”; lo más probable es que simplemente crearon un bot y no pensaron en cómo podía utilizarse de formas erróneas. Quizá no sea poner el listón demasiado alto, pero tenemos que estar de acuerdo en que hay que pensar en la posibilidad del mal uso de las aplicaciones que creamos.

Otro ejemplo es que hubo una época en que Google Photos utilizaba un algoritmo de reconocimiento de imágenes que a veces clasificaba fotos de personas negras como “gorilas”.[6] También en este caso es extremadamente improbable que alguien en Google decidiera explícitamente lanzar esta característica (y mucho menos enfrentarse a la “ética” de la misma). Aquí parece posible que el problema sea alguna combinación de datos mal entrenados, la imprecisión del modelo y el carácter gravemente ofensivo del error (si el modelo hubiera reconocido alguna vez buzones de correo como camiones de bomberos, probablemente a nadie le hubiera importado).

En este caso, la solución es menos obvia: ¿cómo se puede asegurar que el modelo entrenado no hará predicciones que son de algún modo ofensivas? Por supuesto que hay que entrenar (y probar) el modelo con una amplia variedad de entradas, pero ¿es posible asegurarse de que no hay alguna entrada en algún momento que hará que el modelo se comporte de un modo que pueda avergonzarnos? Es un problema complicado (Google parece haberlo resuelto simplemente negándose a predecir “gorila”).

Compromiso entre precisión e imparcialidad

Supongamos que estamos creando un modelo que predice la probabilidad de que las personas realicen una determinada acción, y hacemos un buen trabajo, reflejado en la tabla 26.1.

Tabla 26.1. Un trabajo bastante bueno.

Predicción	Personas	Acciones	%
Improbable	125	25	20 %

Probable	125	75	60 %

De las personas que hemos predicho como improbables de realizar la acción, solo el 20 % de ellas la hacen, y de las personas que hemos predicho como probables de realizarla, solo el 60 % de ellas la hacen. No parece tan terrible.

De hecho, al dividir las predicciones por grupos, descubrimos unas estadísticas sorprendentes (tabla 26.2).

Tabla 26.2. Estadísticas sorprendentes.

Grupo	Predicción	Personas	Acciones	%
A	Improbable	100	20	20 %
A	Probable	25	15	60 %
B	Improbable	25	5	20 %
B	Probable	100	60	60 %

¿Es inexacto el modelo? Los científicos de datos participantes esgrimen distintos argumentos:

Argumento 1:

El modelo clasifica el 80 % del grupo A como “improbable”, pero el 80 % del grupo B como “probable”. Este científico de datos se queja de que el modelo no está tratando los dos grupos del mismo modo en el sentido de que está generando predicciones muy diferentes en ambos.

Argumento 2:

Sin tener en cuenta la membresía de grupo, si predecimos “improbable” tenemos un 20 % de posibilidades de acción, y si predecimos “probable” tenemos un 60 %. Este científico de datos insiste en que el modelo es “preciso” en el sentido de que sus

predicciones parecen significar las mismas cosas sin importar el grupo al que se pertenezca.

Argumento 3:

40/125 = 32 % de las personas del grupo B fueron etiquetadas falsamente como “probable”, mientras que solo 10/125 = 8 % de las personas del grupo A lo fueron. Este científico de datos (que considera una predicción “probable” como algo malo) insiste en que el modelo estigmatiza injustamente al grupo B.

Argumento 4:

20/125 = 16 % de las personas del grupo A fueron etiquetadas falsamente como “improbable”, mientras que solo 5/125 = 4 % de las personas del grupo B lo fueron. Este científico de datos (que considera una predicción “improbable” como algo malo) insiste en que el modelo estigmatiza injustamente al grupo A.

¿Cuál de estos científicos de datos tiene razón? ¿La tiene alguno de ellos? Quizá depende del contexto.

Posiblemente pensemos de una manera si los dos grupos son “hombres” y “mujeres” y de otra si los dos grupos son “usuarios de R” y “usuarios de Python”. ¿O quizá no pensemos así si resulta que los usuarios de Python son mayoritariamente hombres y los de R mujeres?

Posiblemente pensemos de una manera si el modelo está destinado a predecir si un usuario de DataSciencester solicitará un trabajo a través de la bolsa de empleo de DataSciencester y de otra si el modelo está prediciendo si un usuario pasará la entrevista para ello.

Posiblemente la opinión dependa del propio modelo, de las características que tiene en cuenta y de los datos en los que fue entrenado.

En cualquier caso, lo que quiero decir es que “precisión” e “imparcialidad” se pueden compensar (dependiendo, por supuesto, de cómo se definan), pero estos compromisos no siempre tienen soluciones “correctas” obvias.

Colaboración

Los funcionarios del gobierno de un país represivo (según sus estándares) han decidido finalmente permitir a los ciudadanos unirse a DataSciencester. Sin embargo, insisten en que los usuarios de su país no tienen permitido hablar sobre *deep learning*. Es más, quieren que usted les informe del nombre de cualquier usuario que incluso intente buscar información sobre el tema.

¿Es mejor que los científicos de datos de ese país tengan acceso a la DataSciencester con limitación en los temas de discusión (y además vigilada) que usted tendría permitido ofrecerles? ¿O son las restricciones propuestas tan terribles que es mejor que no tengan acceso ninguno?

Capacidad de interpretación

El departamento de Recursos Humanos de DataSciencester le pide que desarrolle un modelo que prediga qué empleados tienen más riesgo de abandonar la compañía, de forma que pueda intervenir e intentar contentarles (la tasa de deserción es un componente importante del artículo "Los 10 mejores lugares de trabajo" en el que su director general aspira a aparecer).

Ha recopilado una selección de datos históricos y está pensando en tres modelos:

- Un árbol de decisión.
- Una red neuronal.
- Un "experto en retener a la gente" muy caro.

Uno de sus científicos de datos insiste en que debe utilizar el modelo que mejor funcione.

Otro insiste en que no utilice el modelo de red neuronal, ya que solo los otros dos pueden explicar sus predicciones, y que solo la explicación de las predicciones podrá ayudar a RR. HH. a instaurar cambios generalizados (a diferencia de las intervenciones puntuales).

Un tercero dice que, mientras el "experto" puede ofrecer una explicación

de sus predicciones, no hay razón para creer su palabra de que describe las razones reales que predijo del modo que lo hizo.

Como con nuestros otros ejemplos, aquí no hay ninguna mejor opción absoluta. En ciertas circunstancias (posiblemente por razones legales o si sus predicciones cambian de algún modo la vida), quizá prefiera un modelo que dé peores resultados, pero cuyas predicciones puedan ser explicadas. En otras, quizá solo quiera el modelo que predice mejor y en otras aun distintas quizá no hay modelo interpretable que dé buenos resultados.

Recomendaciones

Como ya dijimos en el capítulo 24, una aplicación habitual de la ciencia de datos es recomendar cosas a la gente. Cuando alguien ve un vídeo de YouTube, YouTube le recomienda vídeos que podría querer ver a continuación.

YouTube gana dinero con la publicidad y (supuestamente) quiere recomendar vídeos que es más probable que la gente quiera ver, de forma que puedan mostrarles más anuncios. No obstante, resulta que a la gente le gusta ver vídeos sobre teorías conspiranoicas, que tienden a aparecer en las recomendaciones.

> **Nota:** En el momento en que escribí este capítulo, si se buscaba en YouTube "saturno", el tercer resultado era "Algo está ocurriendo en Saturno... ¿LO están ocultando?", que quizá le dé una idea del tipo de vídeos de los que estoy hablando.

¿Tiene YouTube la obligación de no recomendar vídeos conspiranoicos? ¿Incluso aunque eso sea lo que mucha gente parece querer ver?

Un ejemplo distinto es que si entramos en `google.com` (o `bing.com`) y empezamos a escribir algo para buscar, el motor de búsqueda ofrecerá sugerencias para autocompletarla. Estas sugerencias están basadas (al menos en parte) en las búsquedas de otras personas; en particular, si otra gente busca

cosas desagradables, quizá ello se vea reflejado en sus sugerencias.

¿Debería intentar un motor de búsqueda filtrar afirmativamente sugerencias que no le gustan? Google (por las razones que sean) parece tener la intención de no sugerir cosas relacionadas con la religión. Por ejemplo, al escribir “mitt romney m” en Bing, la primera sugerencia es “mitt romney mormons” (que es lo que era de esperar), mientras que Google se niega a ofrecer esa sugerencia.

De hecho, Google filtra explícitamente autosugerencias que considera “ofensivas o despreciativas”[7] (cómo decide lo que es ofensivo o despreciativo no queda nada claro). Sin embargo, a veces la verdad es ofensiva. ¿Es ético proteger a la gente de esas sugerencias? ¿O no lo es? ¿O no es en absoluto una cuestión de ética?

Datos sesgados

En la sección “Vectores de palabras” del capítulo 21 utilizamos una serie de documentos para aprender la incrustación de vectores para palabras. Estos vectores estaban diseñados para exhibir similitud distribucional. Es decir, las palabras que aparecen en contextos similares deberían tener vectores similares. En particular, cualquier sesgo aplicado a los datos de entrenamiento se verá reflejado en los propios vectores de palabras.

Por ejemplo, si nuestros documentos tratan todos sobre lo depravados que son los usuarios de R y lo virtuosos que son los de Python, lo más probable es que el modelo aprenda dichas asociaciones para “Python” y “R”.

Más común es que los vectores de palabras estén basados en cierta combinación de noticias de Google, Wikipedia, libros y páginas web rastreadas, lo que significa que aprenderán sean cuales sean los patrones de distribución que estén presentes en dichas fuentes.

Por ejemplo, si la mayoría de los artículos de prensa sobre ingenieros de software son sobre ingenieros de software masculinos, entonces el vector aprendido para “software” podría acercarse más a vectores de otras palabras “masculinas” que a vectores de palabras “femeninas”.

En este punto, cualquier aplicación creada utilizando estos vectores podría

también exhibir esta cercanía. Dependiendo de la aplicación, puede ser o no un problema. En ese caso hay varias técnicas que se pueden intentar para "eliminar" determinados sesgos, aunque probablemente nunca se lleguen a eliminar del todo. Pero es algo de lo que debemos ser conscientes.

De forma similar, como en el ejemplo de la sección "Crear productos de mala calidad" de este mismo capítulo, si entrenamos un modelo con datos no representativos, hay una elevada probabilidad de que no dé buenos resultados en el mundo real, posiblemente en formas que sean ofensivas o vergonzosas.

Siguiendo distintas líneas de pensamiento, también es posible que los algoritmos creados puedan codificar sesgos reales que ya existen en el mundo. Por ejemplo, el modelo de libertad condicional podría hacer un trabajo perfecto prediciendo qué delincuentes liberados volverían a ser detenidos de nuevo, pero si los mismos detenidos nuevamente son el resultado de procesos sesgados en el mundo real, entonces el modelo podría estar perpetuando el sesgo.

Protección de datos

Ya sabemos mucho sobre los usuarios de DataSciencester. Sabemos qué tecnologías les gustan, quiénes son sus amigos científicos de datos, dónde trabajan, cuánto ganan, cuánto tiempo se pasan en el sitio, en qué ofertas de trabajo hacen clic, etc.

El vicepresidente de Monetización quiere vender estos datos a los anunciantes, que están ansiosos por venderles sus distintas soluciones "Big Data" a los usuarios de nuestra red. El científico jefe quiere compartirlos con investigadores académicos, que están deseando publicar artículos sobre quién llega a convertirse en científico de datos. La vicepresidenta de Campañas Electorales tiene planes para suministrar los datos a los organizadores de campañas políticas, que quieren reclutar sus propias organizaciones de ciencia de datos. Y al vicepresidente de Asuntos Gubernamentales le gustaría usar los datos para responder las preguntas que le hagan los cuerpos de seguridad.

Gracias a un iluminado vicepresidente de Contratos, los usuarios

aceptaron las condiciones del servicio, que garantizan el derecho a hacer lo que le dé la gana con sus datos.

Sin embargo (como ya nos hemos acostumbrado a esperar), varios científicos de datos plantean diversas objeciones a estos distintos usos. Uno piensa que está mal pasar los datos a los anunciantes; a otro le preocupa no poder fiarse de los académicos para que protejan los datos con responsabilidad. Un tercero cree que la compañía debería mantenerse al margen de la política, mientras que el último insiste en que no hay que confiar en la policía y que colaborar con los cuerpos de seguridad dañará a gente inocente.

¿Alguno de estos científicos de datos tiene razón?

En resumen

¡Hay tantas cosas de las que preocuparse! Y hay muchas más que ni hemos mencionado, y todavía más que surgirán en el futuro, pero que nunca nos ocurrirían hoy.

Para saber más

- No falta gente que profesa ideas de importancia sobre la ética de datos. Buscar en Twitter (o en el sitio de noticias que prefiera) es probablemente la mejor manera de estar al día sobre la controversia más reciente acerca de la ética de los datos.
- Si quiere algo un poco más práctico, Mike Loukides, Hillary Mason y DJ Patil han escrito un libro electrónico corto denominado *Ethics and Data Science*, en `https://www.oreilly.com/library/view/ethics-and-data/9781492043898/`, sobre poner en práctica la ética de los datos, que me honra recomendar de parte de Mike, dado que fue la persona que estuvo de acuerdo en publicar este libro allá en 2014 (ejercicio: ¿es esto ético por mi parte?).

[1] https://es.wikipedia.org/wiki/Esc%C3%A1ndalo_Facebook-Cambridge_Analytica.

[2] https://www.nytimes.com/2018/05/24/technology/uber-autonomous-car-ntsb-investigation.html.

[3] https://www.themarshallproject.org/2015/08/04/the-new-science-of-sentencing.

[4] https://moneytrainingclub.com/this-is-how-airlines-are-using-algorithms-to-separate-group-seats-and-force-us-to-pay-more/.

[5] https://es.wikipedia.org/wiki/Tay_(bot).

[6] https://www.theverge.com/2018/1/12/16882408/google-racist-gorillas-photo-recognition-algorithm-ai.

[7] https://blog.google/products/search/google-search-autocomplete/.

27 Sigamos haciendo ciencia de datos

Y ahora, una vez más, permito que mi monstruosa progenie salga a la luz y prospere.

—Mary Shelley

A partir de aquí, ¿cuál es el camino a seguir? Suponiendo que la ciencia de datos no haya ahuyentado a mis lectores, hay una serie de cosas que conviene aprender a continuación.

IPython

Ya mencioné IPython (`http://ipython.org/`) al principio de este libro. Ofrece un intérprete con mucha más funcionalidad que el del Python estándar e incorpora "funciones mágicas" que permiten (entre otras cosas) copiar y pegar código fácilmente (cosa que suele ser complicada por la combinación entre las líneas en blanco y el formato de los espacios) y ejecutar fragmentos de código dentro del intérprete.

Dominar IPython facilita mucho la vida (incluso aprender solo un poquito ya basta para hacernos las cosas más fáciles).

Nota: En la primera edición, también recomendaba aprender IPython (ahora Jupyter) Notebook, un entorno de trabajo interactivo que permite combinar texto, código Python y visualizaciones.
Desde entonces, me he vuelto un escéptico de este entorno (`https://www.youtube.com/watch?v=7jiPeIFXb6U`), porque me parece que confunde a los principiantes y anima a realizar malas prácticas de codificación (y por otras muchas otras razones también). Seguro que muchas personas que no son yo le animarán intensamente a que lo use, pero recuerde que era yo el que estaba en contra.

Matemáticas

A lo largo de este libro hemos hecho nuestros pinitos en el álgebra lineal (capítulo 4), la estadística (capítulo 5), la probabilidad (capítulo 6) y distintos aspectos del *machine learning*.

Para ser un buen científico de datos conviene saber mucho más sobre estos temas, por lo que animo a mis lectores a que estudien más a fondo cada uno de ellos, utilizando los libros recomendados al final de los capítulos, los libros de texto que prefieran, cursos en línea o incluso cursos presenciales.

No desde cero

Implementar cosas "desde cero" es fantástico para entender cómo funcionan. Pero en general no es tan bueno para la facilidad de manejo, la creación rápida de prototipos, el manejo de errores o el rendimiento (a menos que lo hagamos teniendo cualquiera de estas premisas específicamente en mente).

En la práctica, es oportuno utilizar librerías bien diseñadas que desarrollen de manera sólida los fundamentos. En mi propuesta inicial para este libro se incluía una segunda mitad para aprender las librerías que O'Reilly vetó, cosa que debo agradecerles. Desde que se publicó la primera edición, Jake VanderPlas ha escrito *Python Data Science Handbook* (`http://shop.oreilly.com/product/0636920034919.do`), también de O'Reilly, que es una buena introducción a las librerías más importantes y sería un buen libro para leer después de este.

NumPy

NumPy, en `http://www.numpy.org`, que significa "Numeric Python", ofrece servicios para hacer "verdadera" computación científica. Dispone de *arrays* que funcionan mejor que nuestros vectores `list`, matrices que hacen un mejor trabajo que nuestras matrices `list` de `list` y muchas funciones

numéricas para trabajar con todos ellos.

NumPy es básica para muchas otras librerías, por lo cual conocerla resulta especialmente valioso.

pandas

pandas, en `http://pandas.pydata.org`, ofrece estructuras de datos adicionales para trabajar con conjuntos de datos en Python. Su principal abstracción es `DataFrame`, similar en concepto a la clase `Table` de NotQuiteABase que construimos en el capítulo 24, pero con mucha más funcionalidad y mejor rendimiento. Si la idea es utilizar Python para mezclar, dividir, agrupar y manipular conjuntos de datos, pandas es una herramienta de gran valor.

scikit-learn

scikit-learn, en `http://scikit-learn.org`, es probablemente la librería más conocida para hacer *machine learning* en Python. Contiene todos los modelos que hemos implementado y muchos más que no hemos hecho aquí. En un problema real, nunca se construiría un árbol de decisión desde el principio; scikit-learn ya se encarga de ello. Tampoco nadie escribiría un algoritmo de optimización a mano, ya que para eso está scikit-learn, que ya utiliza uno muy bueno. Su documentación contiene muchos ejemplos (`http://scikit-learn.org/stable/auto_examples/`) de lo que puede hacer (y, generalizando, de lo que el *machine learning* puede hacer).

Visualización

Los gráficos matplotlib que hemos estado creando han sido limpios y funcionales, pero no especialmente estilosos (y para nada interactivos). Si la intención es profundizar en la visualización de datos, existen varias opciones.

La primera es explorar más a fondo matplotlib, de la que solo hemos tratado unas cuantas funciones. Su sitio web contiene muchos ejemplos, en

https://matplotlib.org/2.0.2/examples/index.html, de su funcionalidad y una galería, en https://matplotlib.org/stable/gallery/index.html, de algunas de las más interesantes. Si lo que se desea es crear visualizaciones estáticas (por ejemplo, para imprimir en un libro), probablemente este es el mejor siguiente paso.

También es interesante echar un vistazo a seaborn, en https://seaborn.pydata.org/, una librería que (entre otras cosas) hace a matplotlib más atractiva.

Para crear visualizaciones interactivas y compartirlas en la web, la opción obvia es probablemente D3.js, en https://d3js.org, una librería de JavaScript para crear "documentos basados en datos" (o *data-driven documents*, de donde proviene lo de D3). Incluso no sabiendo mucho JavaScript, perfectamente se pueden copiar ejemplos de la galería de D3, en https://github.com/d3/d3/wiki/Gallery, y modificarlos para que trabajen con nuestros datos (los buenos científicos de datos copian de la galería de D3; los grandes científicos de datos roban información de ella).

Aun sin tener interés alguno en D3, solo echando un vistazo a la galería se aprende muchísimo sobre visualización de datos.

Bokeh, en https://docs.bokeh.org/en/latest/, es un proyecto que incorpora funcionalidad de estilo D3 en Python.

R

Aunque se podría pasar perfectamente sin saber R, en http://www.r-project.org, muchos científicos de datos y proyectos de ciencia de datos lo usan, así que merece la pena al menos familiarizarse con ello.

En parte conviene para poder entender los *posts* de blogs basados en R de la gente, y sus ejemplos y códigos; en parte sirve para apreciar más la limpia elegancia de Python (en comparación) y en parte también es bueno para poder ser un participante más informado en las interminables guerras "R contra Python".

Deep learning (aprendizaje profundo)

Se puede ser científico de datos sin tener ni idea de *deep learning*, pero no se puede ser un científico de datos que está a la última sin conocerlo.

Los dos *frameworks* de *deep learning* más conocidos para Python son TensorFlow, en https://www.tensorflow.org/, creado por Google, y PyTorch, en https://www.pytorch.org/, creado por Facebook. Internet está lleno de tutoriales para ellos que van desde maravillosos hasta terribles.

TensorFlow es más antiguo y se utiliza más, pero PyTorch es (en mi opinión) mucho más fácil de manejar y (en particular) mejor para el principiante. Yo prefiero (y recomiendo) PyTorch, pero (como suele decirse) nadie fue nunca despedido por elegir TensorFlow.

Encontrar datos

Si parte de su trabajo es hacer ciencia de datos, probablemente obtendrá los datos como parte de su trabajo (aunque no necesariamente). ¿Y qué pasa si hace ciencia de datos por diversión? Hay datos por todas partes, pero aquí ofrezco algunos puntos de partida:

- Data.gov, en http://www.data.gov, es el portal de datos abierto del gobierno de los Estados Unidos. Si necesita datos sobre cualquier cosa que tenga que ver con el gobierno (que en estos días parece ser casi todo), es un buen lugar para empezar.
- Reddit tiene un par de foros, r/datasets, en http://www.reddit.com/r/datasets, y r/data, en http://www.reddit.com/r/data, que son lugares tanto para pedir como para descubrir datos.
- Amazon.com mantiene una colección de conjuntos de datos públicos, en https://registry.opendata.aws/, que interesa analizar utilizando sus productos (pero que también se pueden analizar con los productos que se deseen).

- Robb Seaton tiene una extravagante lista de conjuntos de datos tratados en su blog, en http://rs.io/100-interesting-data-sets-for-statistics/.
- Kaggle, en http://www.kaggle.com/, es un sitio que alberga competiciones de ciencia de datos. Nunca logré entrar (mi naturaleza competitiva no sale mucho a la luz cuando se trata de ciencia de datos), pero puede probar. Tienen muchos conjuntos de datos.
- Google tiene un nuevo módulo Dataset Search, en https://datasetsearch.research.google.com/, que permite (exacto) buscar conjuntos de datos.

Haga ciencia de datos

Buscar en catálogos de datos está bien, pero los mejores proyectos (y productos) son los que despiertan el interés. Estos son algunos de los que yo he hecho.

Hacker News

Hacker News, en https://news.ycombinator.com/news, es un sitio de debate y agregación para noticias relacionadas con la tecnología. Recopila muchos artículos, muchos de los cuales no me resultan en absoluto interesantes.

En consecuencia, hace varios años me lancé a construir un clasificador de historias de Hacker News, en https://github.com/joelgrus/hackernews, para predecir si yo estaría interesado o no en una determinada historia, lo que no les hizo mucha gracia a los usuarios de Hacker News, que se ofendieron con la idea de que alguien pudiera no estar interesado en todas y cada una de las historias del sitio.

Esto implicaba etiquetar a mano muchas historias (para tener un conjunto de entrenamiento), elegir características de la historia (por ejemplo, palabras del título o dominios de los enlaces) y entrenar un clasificador Naive Bayes

no muy diferente a nuestro filtro de *spam*.

Por razones ahora perdidas en la historia, lo creé en Ruby. Aprenda de mis errores.

Camiones de bomberos

Durante muchos años viví en una calle principal del centro de Seattle, a mitad de camino entre un parque de bomberos y la mayoría de los incendios de la ciudad (o eso parecía). En consecuencia, desarrollé un interés recreativo en el departamento de bomberos de Seattle.

Por suerte (desde el punto de vista de los datos), tienen un sitio web en tiempo real, en `https://sfdlive.com/`, que lista cada alarma de incendios junto con los camiones de bomberos participantes.

Así, para dar rienda suelta a mi interés, extraje muchos años de datos de alarmas de incendios y realicé un análisis de red social, en `https://github.com/joelgrus/fire`, de los camiones de bomberos. Entre otras cosas, ello me obligó a inventar una noción de centralidad específica para camiones de bomberos, a la que llamé TruckRank.

Camisetas

Tengo una hija pequeña, y una incesante fuente de preocupación para mí durante su infancia ha sido que la mayoría de las camisetas para niñas son bastante aburridas, mientras que las camisetas para niños son muy divertidas.

En particular, me quedó claro que había una clara diferencia entre las camisetas destinadas a niños de 1 o 2 años y a niñas de la misma edad. Así que me pregunté a mí mismo si podría entrenar un modelo que reconociera estas diferencias.

Spoiler: lo hice, en `https://github.com/joelgrus/shirts`.

Fue necesario para ello descargar las imágenes de cientos de camisetas, reduciéndolas todas al mismo tamaño, convirtiéndolas en vectores de colores de píxel y utilizando regresión logística para crear un clasificador.

Un método miraba simplemente los colores que estaban presentes en cada camiseta; otro hallaba los primeros 10 componentes principales de los

vectores de las imágenes de las camisetas y clasificaba cada una utilizando sus proyecciones en el espacio de 10 dimensiones ocupado por las “camisetas propias” (figura 27.1).

Figura 27.1. Camisetas propias correspondientes al primer componente principal.

Tuits en un globo terráqueo

Durante muchos años quise crear una visualización de un globo terráqueo giratorio. Durante las elecciones de 2016, creé una pequeña aplicación web, en `https://joelgrus.com/2016/02/27/trump-tweets-on-a-globe-aka-fun-with-d3-socketio-and-the-twitter-api/`, que escuchaba tuits geoetiquetados que coincidían con alguna búsqueda (utilicé “Trump”, ya que aparecía en muchos tuits en aquel momento), los visualicé e hice girar un globo terráqueo en su ubicación cuando aparecían.

Era un proyecto de datos entero de JavaScript, así que quizá le convenga aprender un poco de JavaScript.

¿Y usted?

¿Qué le interesa? ¿Qué cuestiones no le dejan conciliar el sueño por las noches? Busque un conjunto de datos (o extraiga de sitios web) y haga un poco de ciencia de datos.

¡Cuénteme lo que descubra! Puede enviarme mensajes a mi correo electrónico `joelgrus@gmail.com` o encontrarme en Twitter en `@joelgrus`

(https://twitter.com/joelgrus/).

Título de la obra original:
Data Science from Scratch, 2nd Edition

Traductora:
Virginia Aranda González

Responsable editorial:
Víctor Manuel Ruiz Calderón

Adaptación de cubierta:
Celia Antón Santos

Edición en formato digital: 2023

Calle Valentín Beato, 21
28037 Madrid
www.anayamultimedia.es

ISBN ebook: 978-84-415-4731-5