2.ª Edición

O'REILLY® ANAYA MULTIMEDIA

Ciencia de datos desde cero

Principios básicos con Python

Joel Grus

Índice de contenidos

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.

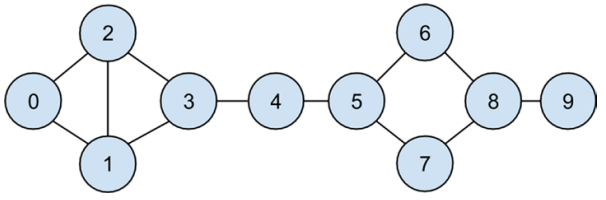

Figura 1.1. La red de DataSciencester.

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 / num_users     # 24 / 10 == 2,4
```

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":

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,

1. `https://matplotlib.org/`.

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()
```

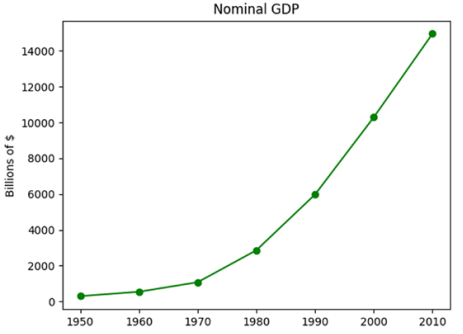

Figura 3.1. Un sencillo gráfico de líneas.

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 histogram.keys()],  # Mueve barras a la derecha en 5
        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
```

```
    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
```

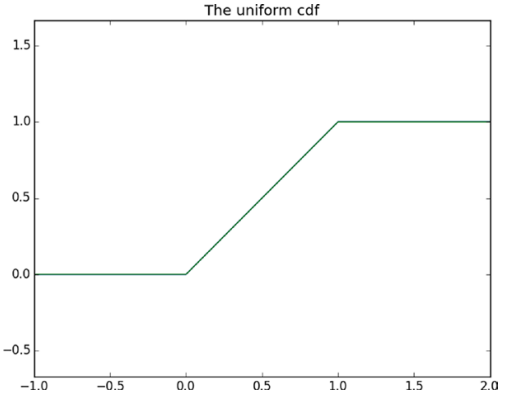

Figura 6.1. La función CDF uniforme.

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.

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.

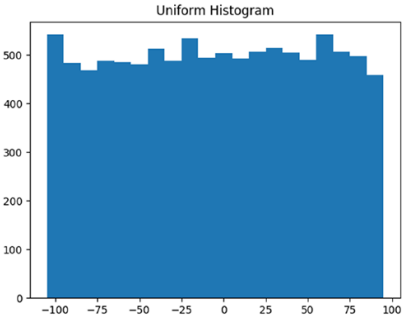

Figura 10.1. Histograma de `uniform`.

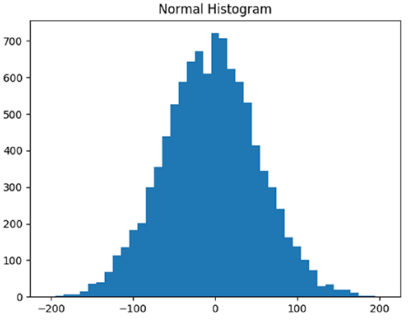

Figura 10.2. Histograma de `normal`.

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.

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:

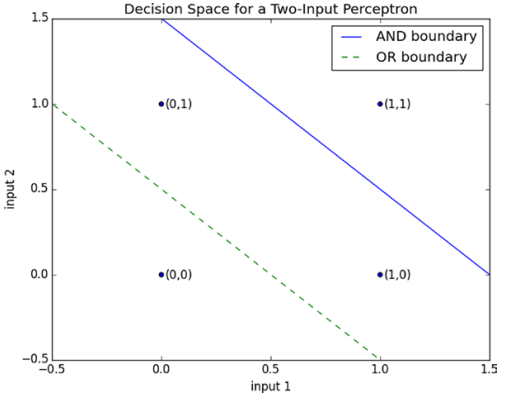

Figura 18.1. Espacio de decisión para un perceptrón de dos entradas.


```
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))
```

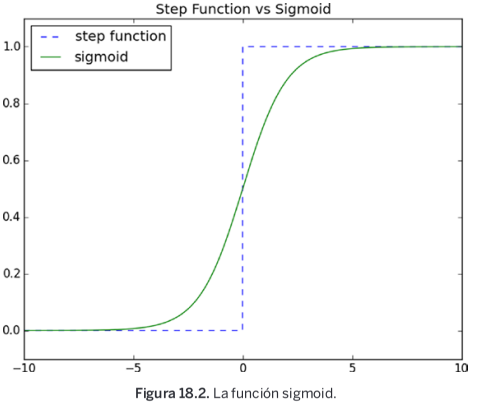

Figura 18.2. La función sigmoid.

¿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, input_with_bias)  # Calcula la entrada
                  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:

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]
```

1. https://es.wikipedia.org/wiki/RGB.

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.

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).

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/`).

Ciencia de datos desde cero

Para aprender de verdad ciencia de datos, no solamente es necesario dominar las herramientas (librerías de ciencia de datos, *frameworks*, módulos y kits de herramientas), sino también conviene comprender las ideas y principio subyacentes. Actualizada para Python 3.6, esta segunda edición de *Ciencia de datos desde cero* muestra cómo funcionan estas herramientas y algoritmos implementándolos desde el principio.

Si ya tiene aptitudes para las matemáticas y ciertas habilidades de programación, el autor Joel Grus, le ayudará a familiarizarse con las mates y las estadísticas, que son el núcleo de la ciencia de datos, y con las habilidades informáticas necesarias para iniciarse como científico de datos. Repleto de nueva información sobre *deep learning*, estadísticas y procesamiento del lenguaje natural, este libro actualizado le muestra cómo sacar lo mejor de la sobreabundancia de datos que actualmente nos rodea.

- Obtenga un curso acelerado de Python.
- Aprenda los fundamentos del álgebra lineal, las estadísticas y la probabilidad (y cómo y cuándo se utilizan en ciencia de datos).
- Recoja, explore, limpie, desmenuce y manipule los datos.
- Bucee en los fundamentos del *machine learning*.
- Implemente modelos como k vecinos más cercanos, Naive Bayes, regresión lineal y logística, árboles de decisión, redes neuronales y agrupamiento (*clustering*).
- Explore los sistemas de recomendación, el procesamiento del lenguaje natural, los análisis de redes, el modelo de programación MapReduce y las bases de datos.

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*.

«Joel te acompaña en un viaje que va desde simplemente alentar tu curiosidad por los datos hasta comprender a fondo los algoritmos básicos que todo científico de datos debe conocer».

—Rohit Sivaprasad
Ingeniero, Facebook

«He recomendado *Ciencia de datos desde cero* a analistas e ingenieros que querían dar el salto al aprendizaje automático. Es la mejor herramienta para comprender los fundamentos de la disciplina».

—Tom Marthaler
Jefe de ingeniería, Amazon

«Traducir conceptos de ciencia de datos a código es difícil. El libro de Joel lo hace mucho más fácil».

—William Cox
Ingeniero de aprendizaje automático, Grubhub

ANAYA MULTIMEDIA
O'REILLY®
www.anayamultimedia.es
ISBN 978-84-415-4720-9
2315184