

2.ª Edición

O'REILLY® ANAYA  
MULTIMEDIA

# Ciencia de datos desde cero

Principios básicos con Python



Joel Grus

# 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

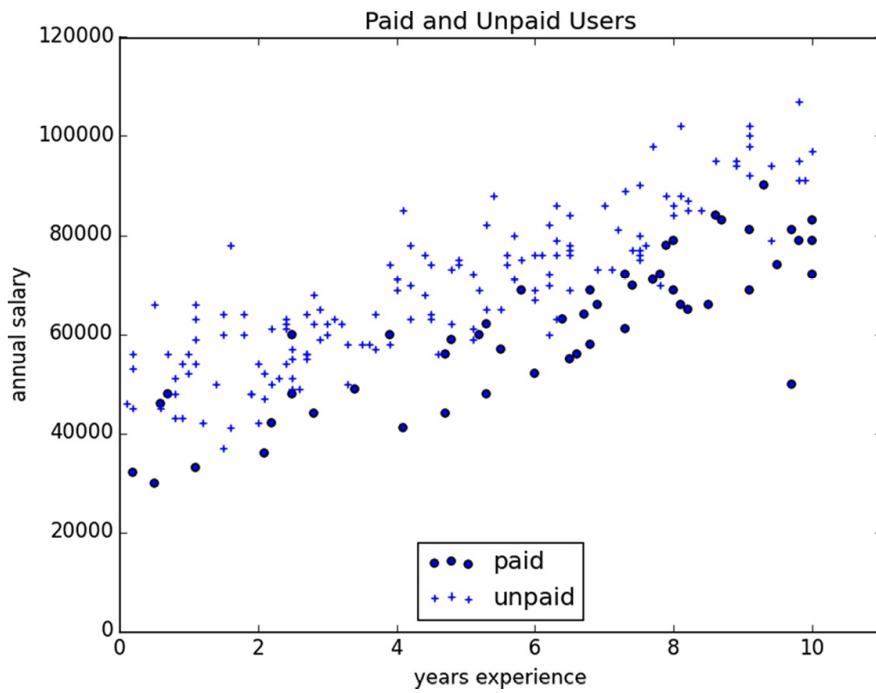
Tenemos un conjunto de datos anónimo de unos 200 usuarios, que contiene el salario de cada usuario, sus años de experiencia como científico de datos y si pagó por una cuenta *premium* (figura 16.1). Como es habitual con las variables categóricas, representamos la variable dependiente como 0 (sin cuenta *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
```

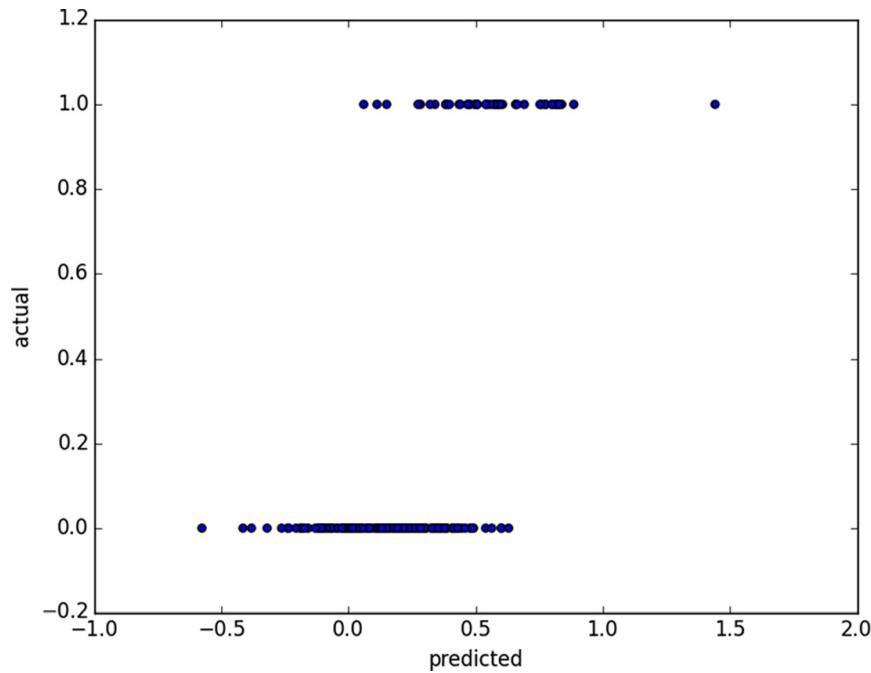
Un primer intento obvio es utilizar regresión lineal y hallar el mejor modelo:

$$\text{cuenta de pago} = \beta_0 + \beta_1 \text{experiencia} + \beta_2 \text{salario} + \varepsilon$$



**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.

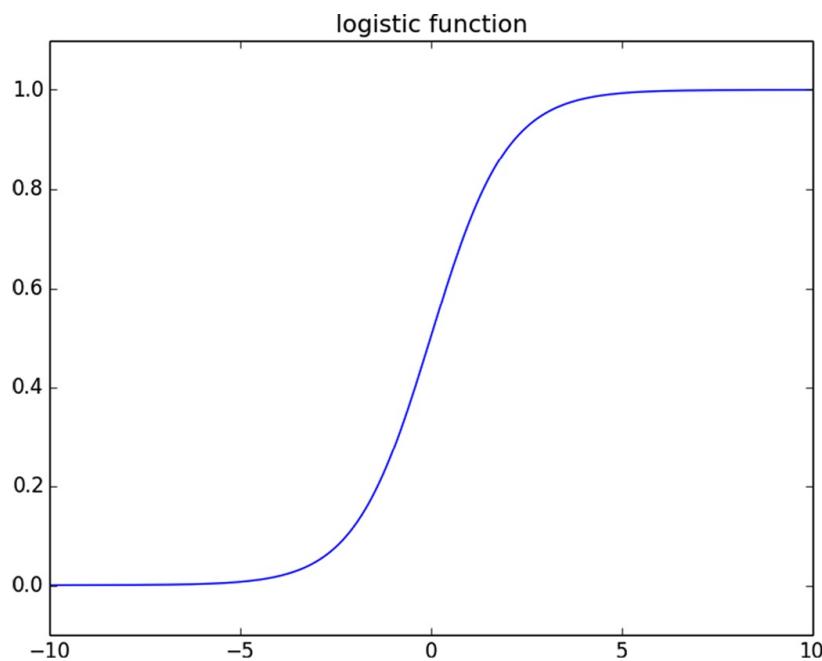
Lo que realmente nos gustaría es que valores grandes positivos de `dot(x_i, beta)` correspondieran a probabilidades cercanas a 1, y que valores grandes negativos correspondieran a probabilidades cercanas a 0. Podemos

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:

$$y_i = f(x_i\beta) + \varepsilon_i$$

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  $\beta$  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  $\beta$ , nuestro modelo dice que cada  $y_i$  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  $y_i$  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:

$$\log L(\beta | x_i, y_i) = y_i \log f(x_i\beta) + (1 - y_i) \log (1 - f(x_i\beta))$$

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 >= 0.5:           # TP: de pago y predecimos de pago
        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

```

```

        false_positives += 1
    else:                                # TN: no de pago y predecimos no de
        true_negatives += 1
precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)

```

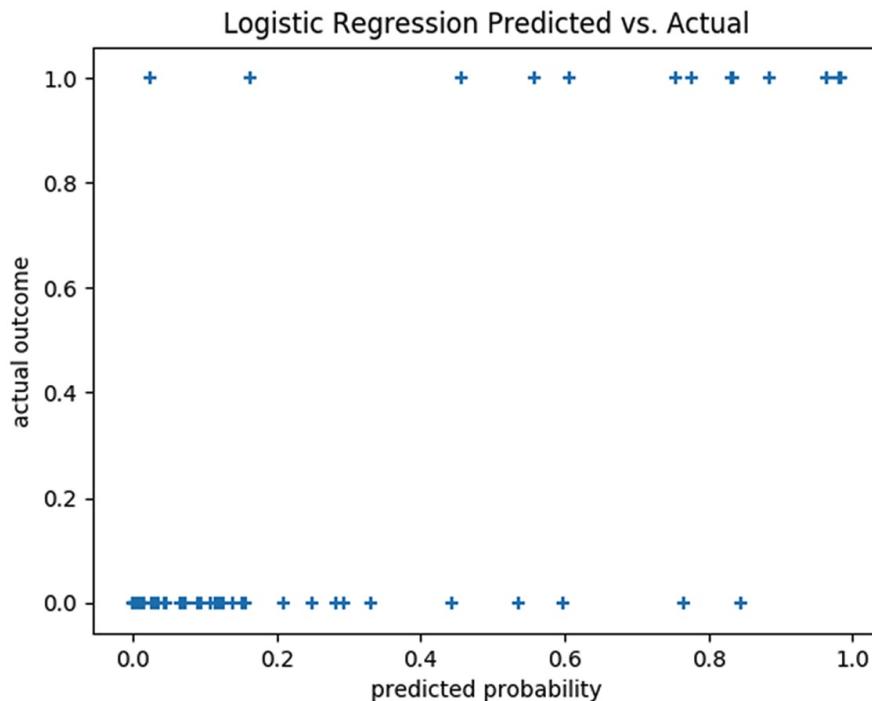
Esto ofrece una precisión del 75 % (“cuando predecimos cuenta de pago acertamos el 75 % de las veces”) y un recuerdo del 80 % (“cuando un usuario tiene una cuenta de pago predecimos cuenta de pago el 80 % de las veces”), lo que no está nada mal, teniendo en cuenta los pocos datos de que disponemos.

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()

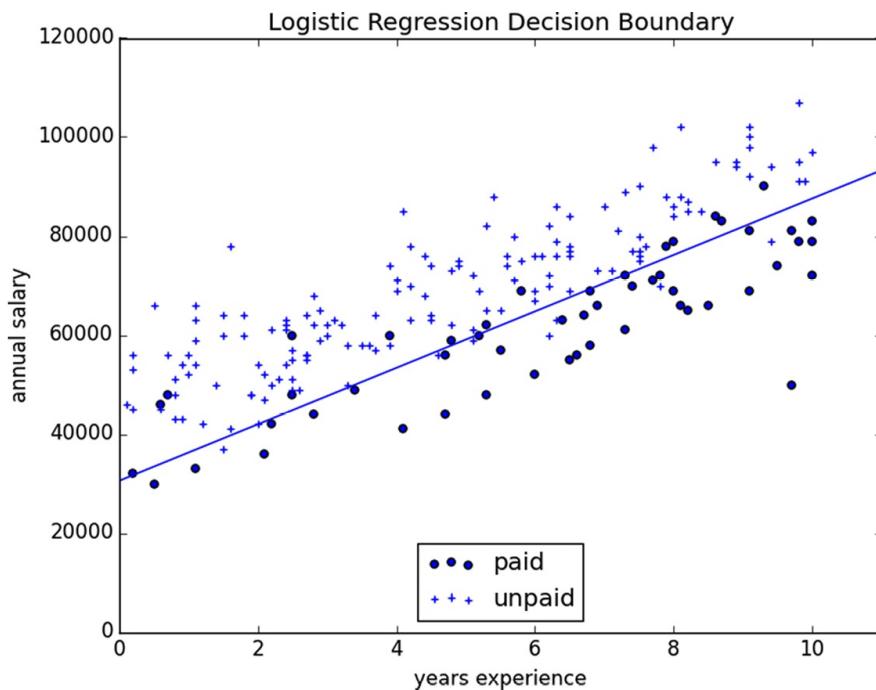
```



**Figura 16.4.** Regresión logística predicha frente a real.

## Máquinas de vectores de soporte

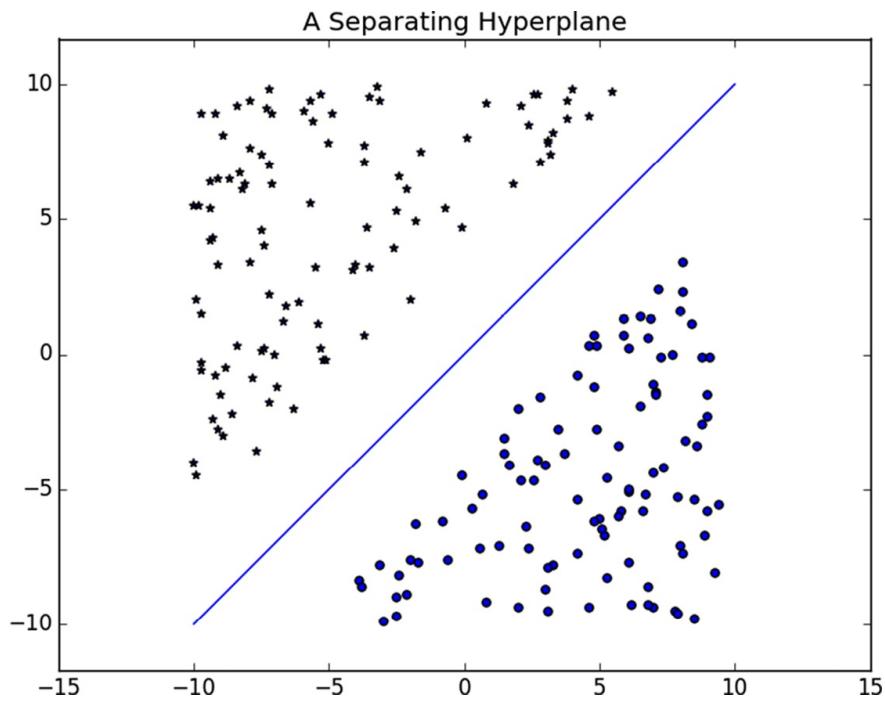
El conjunto de puntos donde  $\text{dot}(\beta, x_i)$  es igual a 0 es la frontera entre nuestras clases. Podemos representar esto para ver exactamente lo que está haciendo nuestro modelo (figura 16.5).



**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).

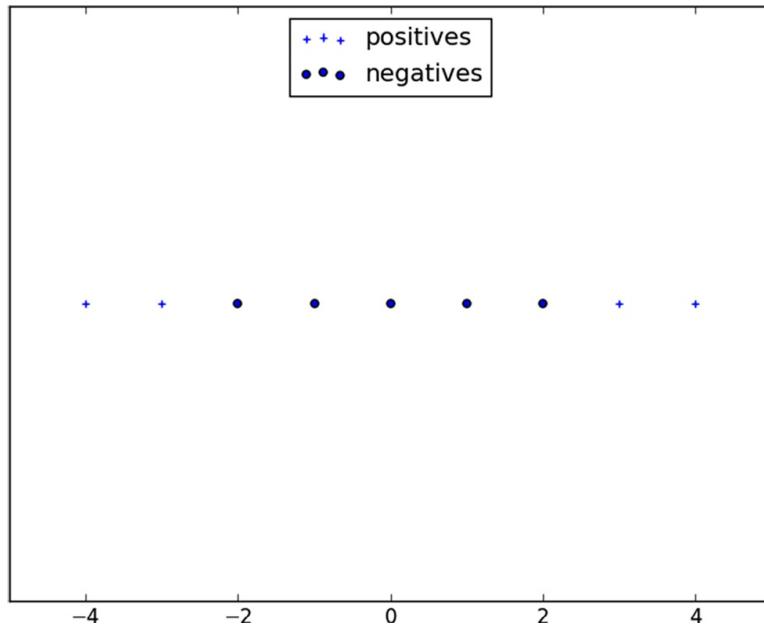


**Figura 16.6.** Un hiperplano de separación.

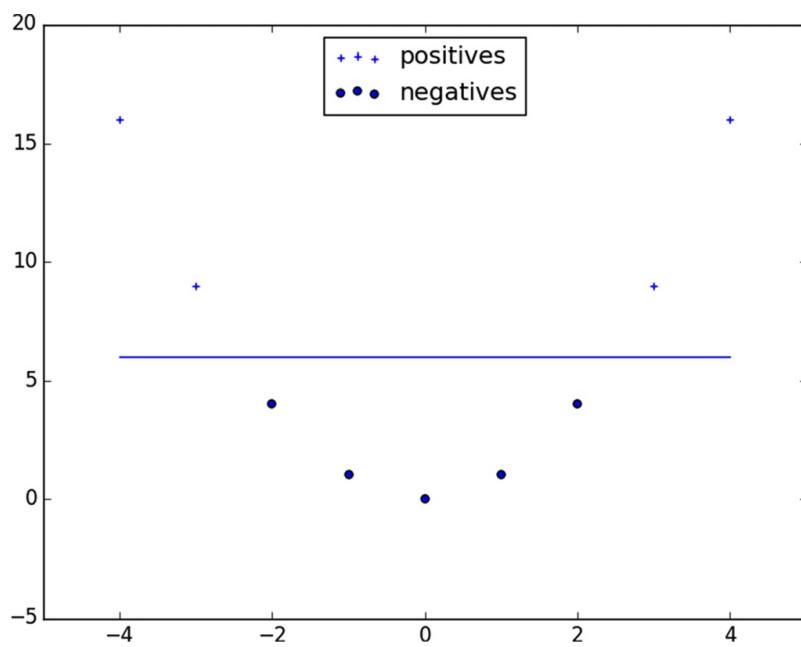
Encontrar un hiperplano como este es un problema de optimización que implica técnicas demasiado avanzadas para nosotros. Un problema distinto es que un hiperplano de separación podría no existir. En nuestro conjunto de datos “¿quién paga?”, simplemente es que no hay línea que separe perfectamente los 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](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 incomprendible.*

—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.

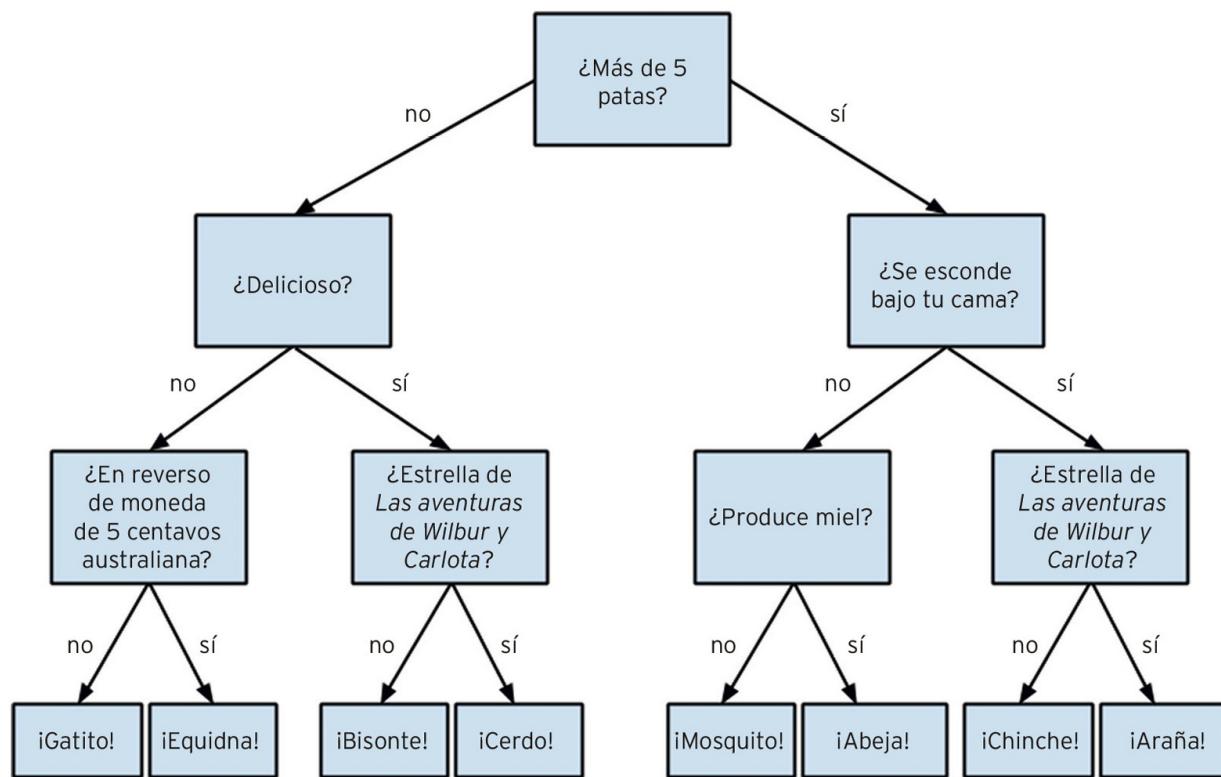
Si alguna vez ha jugado al juego de las 20 preguntas,<sup>1</sup> estará familiarizado con los árboles de decisión. Por ejemplo:

- “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 numéricicos (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.

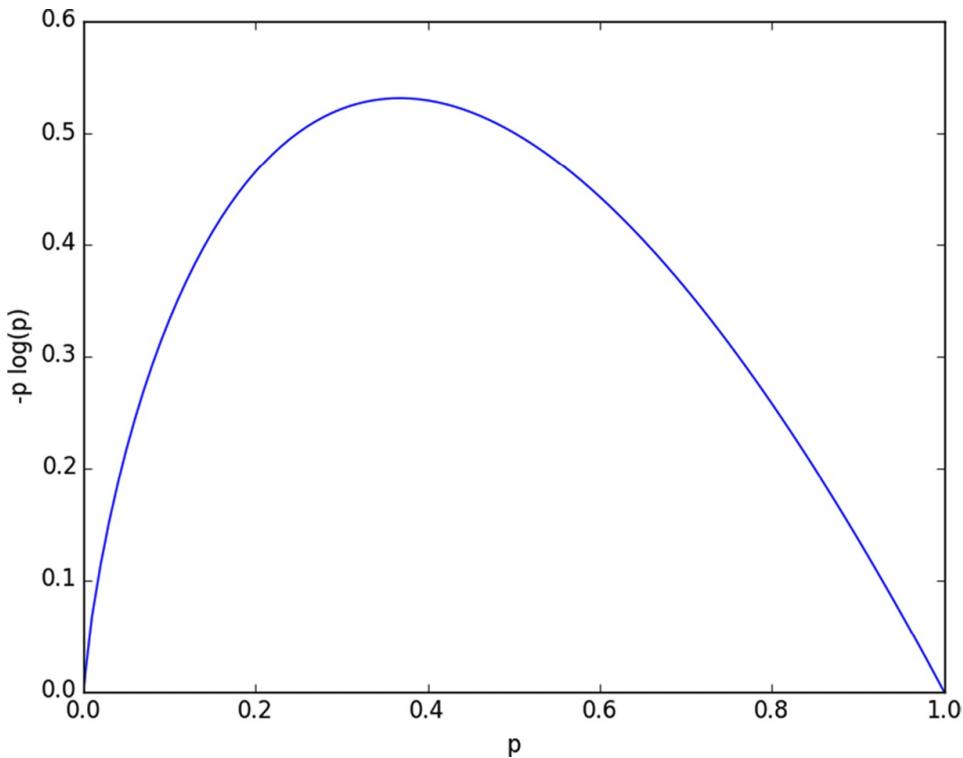
Supongamos que tenemos un conjunto  $S$  de datos, cada miembro del cual está etiquetado como perteneciente a una de un número finito de clases  $C_1, \dots, C_n$ . Si todos los puntos de datos pertenecen a una sola clase, entonces no hay incertidumbre, con lo cual idealmente la entropía sería baja. 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:

$$H(S) = -p_1 \log_2 p_1 - \dots - p_n \log_2 p_n$$

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  $S_1 = \{\text{equidna}\}$  y  $S_2 = \{\text{los demás}\}$ , 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  $S_1, \dots, S_m$  conteniendo proporciones  $q_1, \dots, q_m$  de los datos, calculamos entonces la entropía de la partición como una suma ponderada:

$$H = q_1 H(S_1) + \dots + q_m H(S_m)$$

Que podemos implementar como:

```
def partition_entropy(subsets: List[List[Any]]) -> float:  
    """Returns the entropy from this partition of data into subsets"""\n    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, attribute)           # valor del atributo especificado
        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
```

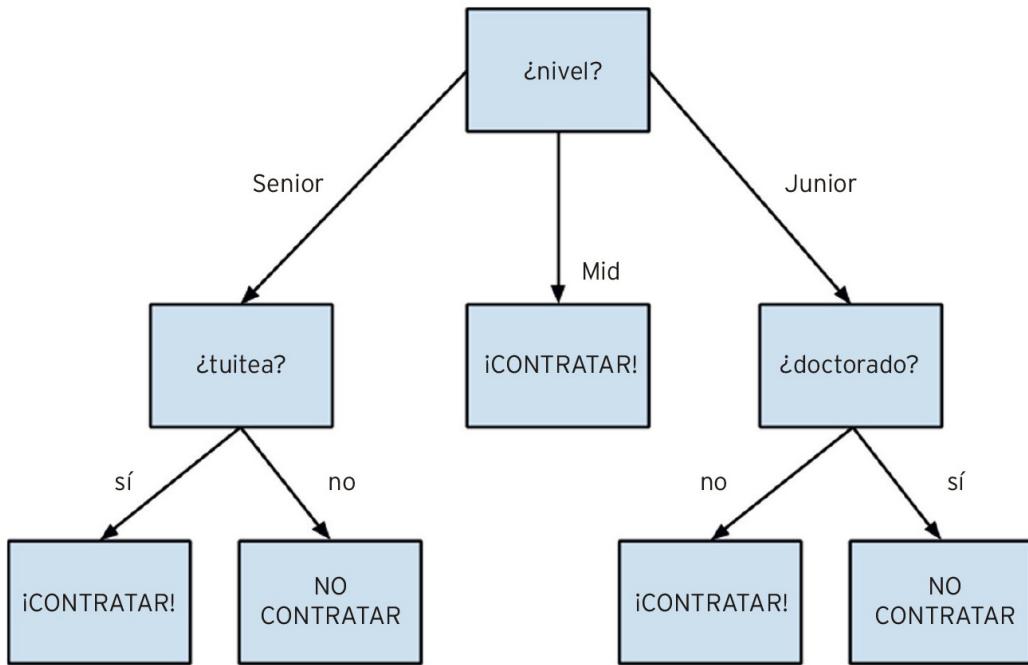
La entropía más baja viene de dividir según `level`, de modo que nos hará falta hacer un subárbol para cada posible valor de `level`. Cada candidato `Mid` se etiqueta `True`, lo que significa que el subárbol `Mid` es simplemente un nodo de hoja que predice `True`. Para candidatos `Senior`, tenemos una 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 repartiéndolo 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:

```

hirинг_tree = Split('level', {
    'Junior': Split('phd', {
        False: Leaf(True),
        True: Leaf(False)
    }),
    'Mid': Leaf(True),
    'Senior': Split('tweets', {
        False: Leaf(False),
        True: Leaf(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 tree.subtrees:
        return tree.default_value
    subtree =
    tree.subtrees[subtree_key]
    return classify(subtree, input)

```

# devuelve el valor predeterminado.  
# Elige el subárbol adecuado  
# 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](https://es.wikipedia.org/wiki/Aprendizaje_basado_en_%C3%A1rb) es un buen punto de partida para un estudio más detallado.

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Twenty\\_questions](https://en.wikipedia.org/wiki/Twenty_questions).