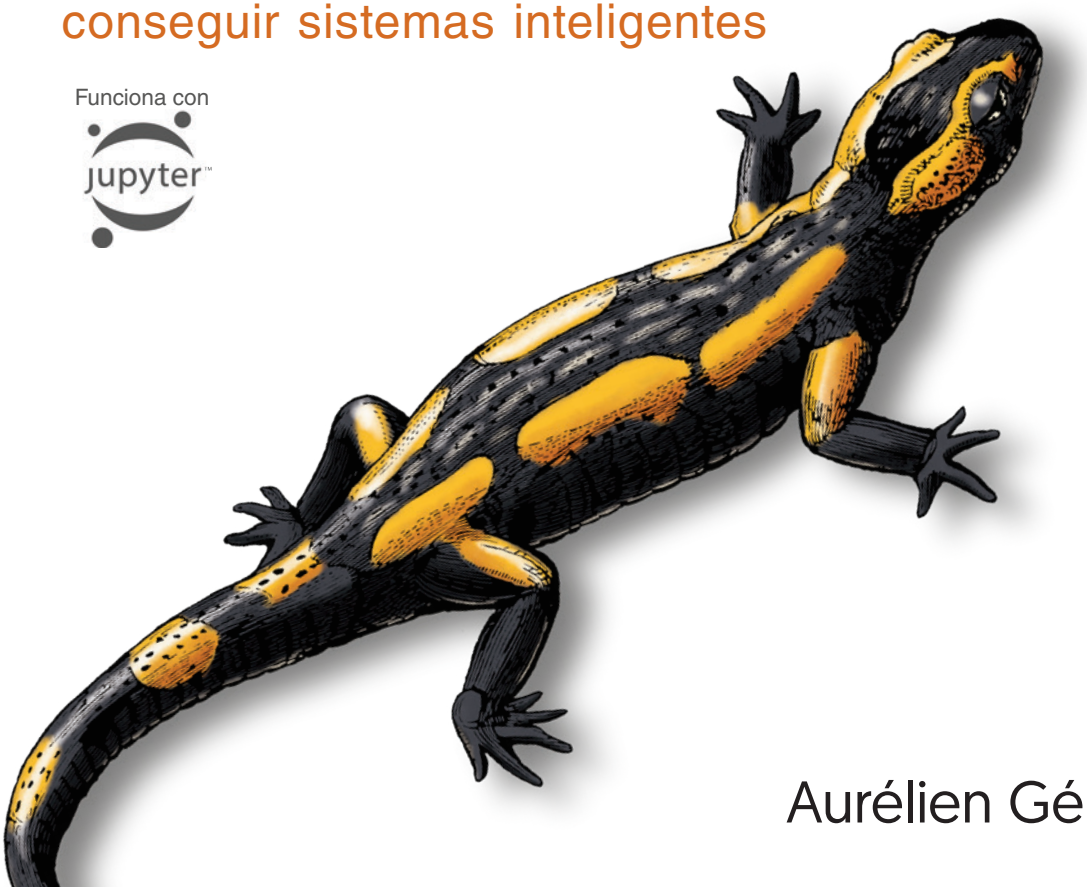
O'REILLY® ANAYA MULTIMEDIA

TERCERA EDICIÓN

Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow

Conceptos, herramientas y técnicas para conseguir sistemas inteligentes

Funciona con jupyter™



Aurélien Géron

Índice de contenidos

1 El paisaje del *machine learning*

No hace tanto tiempo, si hubieses cogido tu teléfono y le hubieses preguntado por dónde ir a casa, te habría ignorado (y la gente habría puesto en duda tu cordura). Pero el *machine learning* ya no es ciencia ficción; miles de millones de personas lo usan a diario. La verdad es que lleva décadas en aplicaciones especializadas, como el reconocimiento óptico de caracteres (*Optical Character Recognition*, OCR). La primera aplicación de *machine learning* que se volvió realmente popular, mejorando las vidas de cientos de millones de personas, ya se apoderó del mundo en los 90: el filtro de *spam*. No es que sea exactamente un robot consciente de sí mismo, pero técnicamente puede considerarse *machine learning*: en realidad, ha aprendido tan bien que ya casi nunca necesitamos marcar un correo como *spam*. Le siguieron cientos de aplicaciones de *machine learning* que hoy en día controlan cientos de productos y características que utilizamos con regularidad: mensajes reproducidos por respuesta de voz interactiva, traducción automática, búsqueda de imágenes, recomendaciones de productos y mucho más.

¿Dónde empieza el *machine learning* y dónde acaba? ¿Qué significa exactamente para una máquina "aprender" algo? Si descargo una copia de todos los artículos de Wikipedia, ¿ha aprendido algo mi ordenador? ¿Es más inteligente de repente? En este capítulo, vamos a empezar por aclarar qué es el *machine learning* y por qué puede interesarnos utilizarlo.

Después, antes de ponernos a explorar el continente del *machine learning*, echaremos un vistazo al mapa y aprenderemos cuáles son las regiones principales y los puntos de referencia más notables: aprendizaje supervisado frente a aprendizaje no supervisado y sus variantes, aprendizaje *online* frente a aprendizaje por lotes, aprendizaje basado en instancias frente a aprendizaje basado en modelos. A continuación, echaremos un vistazo al flujo de trabajo de un proyecto de *machine learning* típico, hablaremos de los principales retos con los que puedes encontrarte y nos ocuparemos de cómo evaluar y perfeccionar un sistema de *machine learning*.

Este capítulo introduce un montón de conceptos fundamentales (y jerga) que todo científico de datos debería saberse de memoria. Será una visión general de alto nivel (es el único capítulo sin mucho código), todo bastante simple, pero mi objetivo es garantizar que todo está claro como el agua antes de que sigamos con el resto del libro. ¡Coge un café y vamos a empezar!

> **Truco:** Si ya estás familiarizado con los conceptos básicos del *machine learning*, puede que te interese pasar directamente al capítulo 2. Si no estás seguro, prueba a contestar todas las preguntas enumeradas al final de este capítulo antes de avanzar.

¿Qué es el *machine learning*?

El *machine learning* es la ciencia (y el arte) de programar ordenadores para que aprendan a partir de datos.

Vamos a ver una definición un poco más general:

> [El *machine learning* es el] campo de estudio que da a los ordenadores la capacidad de aprender sin ser programados de manera explícita.
>
> —Arthur Samuel, 1959

Y una más orientada a la ingeniería:

> Se dice que un programa de ordenador aprende de la experiencia E, con respecto a una tarea T y una medida de rendimiento R, si su rendimiento en T, medido por P, mejora con la experiencia E.
>
> —Tom Mitchell, 1997

El filtro de *spam* es un programa de *machine learning* que, al recibir ejemplos de correo basura (marcados por los usuarios) y ejemplos de correos corrientes (que no sean *spam*, también llamados "*ham*"), puede aprender a marcar el *spam*. Los ejemplos que el sistema utiliza para aprender se llaman conjunto de entrenamiento. Cada ejemplo de entrenamiento se llama instancia de entrenamiento (o muestra). La parte de un sistema de *machine learning* que aprende y realiza predicciones se denomina modelo. Las redes neuronales y los *random forests* son ejemplo de modelos.

En este caso, la tarea T es marcar el *spam* para los correos nuevos, la experiencia E son los datos de entrenamiento y la medida del rendimiento tiene que definirse; por ejemplo, podemos utilizar la proporción de correos clasificados correctamente. Esta medida de rendimiento particular se llama exactitud y se utiliza a menudo en tareas de clasificación.

Si descargamos una copia de todos los artículos de Wikipedia, sin más, nuestro ordenador tiene muchos más datos, pero no es de repente mejor en ninguna tarea. Eso no es *machine learning*.

¿Por qué utilizar *machine learning*?

Piensa en cómo escribirías un filtro de *spam* utilizando técnicas de programación tradicionales (véase la figura 1.1):

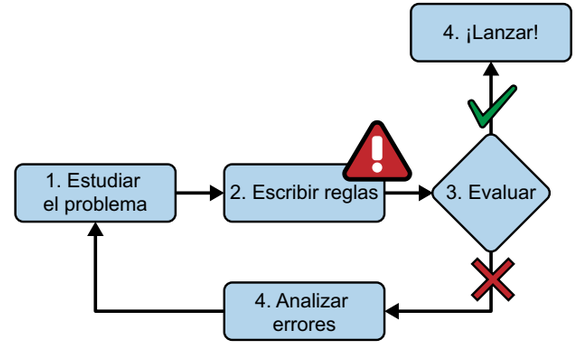


Figura 1.1. El enfoque tradicional.

1. Primero, examinarías cuál es el aspecto típico del *spam*. Podrías fijarte en que algunas palabras o frases (como "Para usted", "tarjeta de crédito", "gratis" e "impresionante") tienden a aparecer mucho en el asunto. Quizá también notarías otros patrones en el nombre del remitente, el cuerpo del correo y en otras partes del mismo.
2. Escribirías un algoritmo de detección para cada uno de los patrones observados y el programa marcaría los correos como *spam* si se detectase una cantidad de esos patrones.
3. Probarías el programa y repetirías los pasos 1 y 2 hasta que fuese lo bastante bueno para lanzarse.

Puesto que el problema es difícil, es probable que el programa se convierta en una lista larga de reglas complejas; sería bastante difícil de mantener.

Por el contrario, un filtro de *spam* basado en *machine learning* aprende de manera automática qué palabras y frases son buenas predictoras de *spam* al detectar patrones de palabras atípicamente frecuentes en ejemplos de *spam* comparados con ejemplos de *ham* (véase la figura 1.2). El programa es mucho más corto, más fácil de mantener y, probablemente, más exacto.

¿Qué pasa si las personas que envían *spam* se dan cuenta de que todos sus correos que contienen "Para usted" se bloquean? Podrían empezar a escribir "Para ti" en su lugar. Un filtro de *spam* que utilice técnicas de programación tradicionales tendría que actualizarse para marcar los correos que tuviesen "Para ti". Si la gente que envía *spam* sigue encontrando maneras de evitar el filtro, tendrás que seguir escribiendo reglas nuevas para siempre.

Por el contrario, un filtro de *spam* basado en técnicas de *machine learning* se da cuenta de forma automática de que "Para ti" se ha vuelto atípicamente frecuente en los correos marcados como *spam* por los usuarios y empieza a marcarlos sin nuestra intervención (véase la figura 1.3).

- PapersWithCode.com (`https://paperswithcode.com/datasets`).
- C Irvine Machine Learning Repository (`https://archive.ics.uci.edu/ml`).
- Conjuntos de datos AWS de Amazon (`https://registry.opendata.aws/`).
- Conjuntos de datos de TensorFlow (`https://tensorflow.org/datasets`).

- Portales meta (ofrecen listas de repositorios de datos abiertos):
  - DataPortals.org (`https://dataportals.org`).
  - OpenDataMonitor.eu (`https://opendatamonitor.eu`).
- Otras páginas que ofrecen listas con muchos repositorios de datos abiertos populares:
  - Lista de conjuntos de datos para *machine learning* de Wikipedia (`https://homl.info/9`).
  - Quora.com (`https://homl.info/10`).
  - El *subreddit* de conjuntos de datos (`https://www.reddit.com/r/datasets`).

En este ejemplo, vamos a utilizar el conjunto de datos de los precios de las casas de California (*California Housing Prices*) del repositorio StatLib[1] (véase la figura 2.1). Este conjunto de datos se basa en los datos del censo de California de 1990. No es exactamente reciente (en aquella época, una casa bonita en el Área de la Bahía aún era asequible), pero tienen muchas cualidades para el aprendizaje, así que vamos a fingir que son datos recientes. Para fines didácticos, he añadido un atributo categórico y he eliminado algunas características.

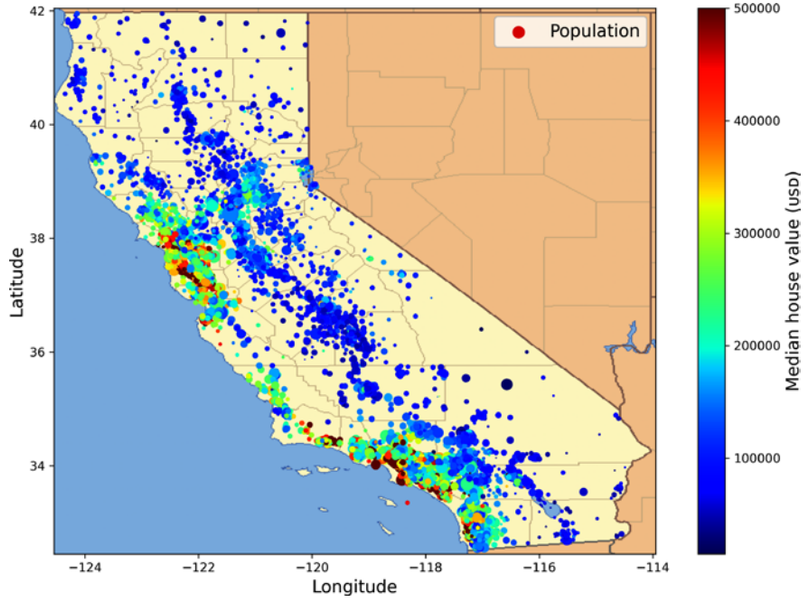


Figura 2.1. Precios de las casas de California.

1. El conjunto de datos original apareció en la publicación de R. Kelley Pace y Ronald Barry, "Sparse Spatial Autoregressions", en *Statistics & Probability Letters 33*, n.º 3 (1997): 291-297.

Tener una visión amplia

¡Bienvenido al Departamento de vivienda del *machine learning*! Tu primera tarea será utilizar el censo de California para crear un modelo de precios de casas en el estado. Estos datos incluyen métricas como la población, los ingresos medianos y los precios medianos de las casas para cada grupo de bloques en California. Los grupos de bloques son la unidad geográfica más pequeña para la que la Oficina del Censo de EE. UU. publica datos simples (un grupo de bloques suele tener una población de entre 600 y 3.000 personas). Por acortar, vamos a llamarlos "distritos".

El modelo debería aprender a partir de estos datos y ser capaz de predecir el precio mediano de las casas en cualquier distrito si se le dan todas las demás métricas.

> **Truco:** Como eres un científico de datos bien organizado, lo primero que deberías hacer es sacar tu lista de comprobación de proyectos de *machine learning*. Puedes empezar con la que aparece en el apéndice A; debería servir bastante bien para la mayoría de proyectos de *machine learning*, pero asegúrate de adaptarla a tus necesidades. En este capítulo, pasaremos por muchos puntos de la lista, pero también nos saltaremos algunos, bien porque son evidentes, bien porque los trataremos en capítulos posteriores.

Enmarcar el problema

Lo primero que debes preguntar a tu jefe es cuál es exactamente el objetivo empresarial. Es probable que crear un modelo no sea la meta final. ¿Cómo espera la empresa utilizar ese modelo y beneficiarse de él? Conocer el objetivo es importante porque determinará la manera de enmarcar el problema, qué algoritmos seleccionar, qué medida de rendimiento utilizar para evaluar el modelo y cuánto esfuerzo se dedicará a ajustarlo.

El jefe responde que la salida del modelo (una predicción del precio mediano de las casas de un distrito) se introducirá en otro sistema de *machine learning* (véase la figura 2.2), junto con muchas otras señales.[2] Este sistema descendente determinará si merece la pena invertir en un área determinada. Hacer bien esto es fundamental, ya que afecta de forma directa a los ingresos.

La siguiente pregunta que debes hacer a tu jefe es cómo es la solución actual (si la hay). A menudo, la situación actual te dará una referencia para el rendimiento, además de perspectivas sobre cómo resolver el problema. Tu jefe responde que actualmente los precios de las casas del distrito los estiman a mano un grupo de expertos: un equipo reúne información actualizada sobre un distrito y, cuando no pueden obtener el precio mediano de las casas, lo calculan mediante reglas complejas.

2. Una información introducida en un sistema de *machine learning* suele llamarse "señal", en referencia a la teoría de la información de Claude Shannon, que desarrolló en Bell Labs para mejorar las telecomunicaciones. Su teoría: nos conviene una relación señal/ruido alta.

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)  # == 3530 / (687 + 3530)
0.8370879772350012
>>> recall_score(y_train_5, y_train_pred)  # == 3530 / (1891 + 3530)
0.6511713705958311
```

Ahora nuestro detector de cincos ya no parece tan brillante como cuando nos fijábamos en la exactitud.

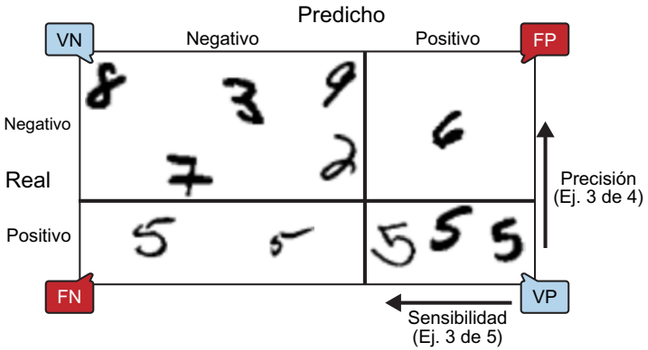


Figura 3.3. Una matriz de confusión ilustrada muestra ejemplo de verdaderos negativos (arriba, izquierda), falsos positivos (arriba, derecha), falsos negativos (abajo, izquierda) y verdaderos positivos (abajo derecha).

Cuando afirma que una imagen representa un 5, solo está en lo cierto el 83,7 % de las veces. Además, solo detecta el 65,1 % de los cincos.

A menudo, es conveniente combinar precisión y sensibilidad en una sola métrica llamada valor F_1, sobre todo cuando necesitas una sola métrica para comparar dos clasificadores. El valor F_1 es la media armónica de la precisión y la sensibilidad (véase la ecuación 3.3). Mientras que la media regular trata a todos los valores igual, la media armónica da mucho más peso a los valores bajos. Como resultado, el clasificador solo obtendrá un valor F_1 si tanto la sensibilidad como la precisión son altas.

Ecuación 3.3. F_1.

$$F_1 = \frac{2}{\frac{1}{\text{precisión}} + \frac{1}{\text{sensibilidad}}} = 2 \times \frac{\text{precisión} \times \text{sensibilidad}}{\text{precisión} + \text{sensibilidad}} = \frac{VP}{VP + \frac{FN + FP}{2}}$$

Para calcular el valor F_1, solo tienes que llamar a la función `f1_score()`:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7325171197343846
```

El valor F_1 prefiere los clasificadores que tienen una precisión y una sensibilidad similares. Eso no siempre es lo que queremos: en algunos contextos, nos interesa más la precisión, mientras que, en otros, tenemos más interés en la sensibilidad. Por ejemplo, si hemos entrenado un clasificador para detectar vídeos que sean seguros para niños, es probable que prefiramos un clasificador que rechace muchos vídeos buenos (baja sensibilidad), pero que mantenga solo los que son seguros (alta precisión), en vez de un clasificador que tenga una sensibilidad mucho más alta pero que permita que aparezcan unos pocos vídeos muy malos en el producto (en casos así, te convendría añadir una *pipeline* humana para comprobar la selección de vídeos del clasificador).

Por otra parte, supongamos que entrenas un clasificador para detectar ladrones de tiendas en imágenes de vigilancia: es probable que esté bien que solo tenga un 30 % de precisión, siempre y cuando tenga una sensibilidad del 99 % (sí, vale, los guardias de seguridad recibirán alguna alerta falsa, pero se atrapará a casi todos los ladrones). Por desgracia, no podemos tener ambas cosas: aumentar la precisión reduce la sensibilidad y viceversa. Esto se denomina compensación precisión/sensibilidad.

Compensación precisión/sensibilidad

Para entender esta compensación, vamos a fijarnos en cómo toma `SGDClassifier` sus decisiones. Para cada instancia, calcula una puntuación basándose en una función de decisión.

Si esa puntuación es mayor que un umbral, asigna la instancia a la clase positiva; de lo contrario, la asigna a la clase negativa. La figura 3.4 muestra unos dígitos posicionados desde la puntuación más baja en la izquierda a la puntuación más alta en la derecha. Supongamos que el umbral de decisión se sitúa en la flecha central (entre los dos cincos): encontrarás 4 verdaderos positivos (cincos reales) a la derecha de ese umbral y un falso positivo (un 6, en realidad). Por tanto, con ese umbral, la precisión es del 80 % (4 de 5). Pero de 6 cincos reales, el clasificador solo detecta 4, así que la sensibilidad es del 67 % (4 de 6). Si elevas el umbral (al moverlo a la flecha de la derecha), el falso positivo (el 6) se convierte en un verdadero negativo; de ese modo se incrementa la precisión (hasta el 100 %, en este caso), pero un verdadero positivo se convierte en un falso negativo, lo que reduce la sensibilidad al 50 %. A la inversa, bajar el umbral incrementa la sensibilidad y reduce la precisión.

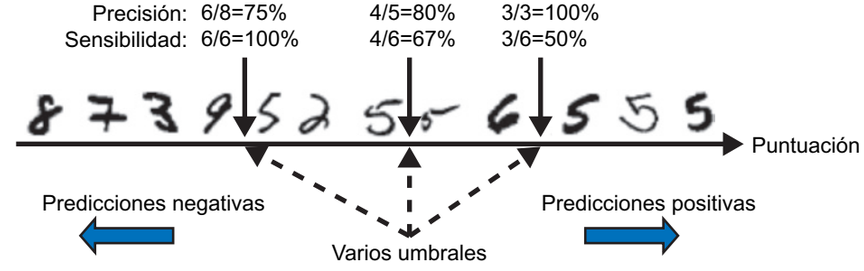


Figura 3.4. La compensación precisión/sensibilidad: las imágenes se ordenan por la puntuación del clasificador y las que están por encima del umbral de decisión se consideran positivos; cuanto más arriba esté el umbral, menor será la sensibilidad, pero (en general) mayor será la precisión.

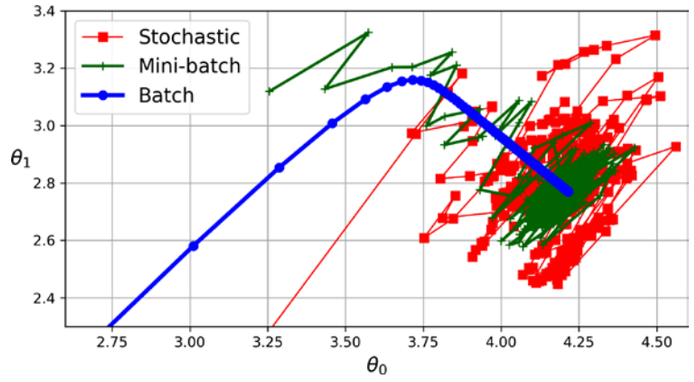


Figura 4.11. Rutas de los descensos de gradiente en el espacio de parámetros.

Tabla 4.1. Comparación de algoritmos para regresión lineal.

Algoritmo	*m* grande	Soporte *out of core*	*n* grande	Hiperparám.	Escalado requerido	Scikit-Learn
Ecuación normal	Rápido	No	Lento	0	No	N/D
SVD	Rápido	No	Lento	0	No	`LinearRegression`
DG por lotes	Lento	No	Rápido	2	Sí	`SGDRegressor`
DG estocástico	Rápido	Sí	Rápido	≥2	Sí	`SGDRegressor`
DG por minilotes	Rápido	Sí	Rápido	≥2	Sí	N/D

Casi no hay diferencia después del entrenamiento: todos estos algoritmos acaban con modelos muy similares y hacen predicciones exactamente de la misma manera.

Regresión polinomial

¿Qué pasa si tus datos son más complejos que una línea recta? Sorprendentemente, puedes utilizar un modelo lineal para que se ajuste a datos no lineales. Una manera sencilla de hacerlo es añadir potencias a cada característica como características nuevas y después entrenar un modelo lineal en este conjunto ampliado de características. Esta técnica se llama "regresión polinomial". Vamos a ver un ejemplo. Primero, generamos datos no lineales (véase la figura 4.12), basados en una ecuación de segundo grado simple (que es una ecuación con la forma $y = ax^2 + bx + c$) y algo de ruido:

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

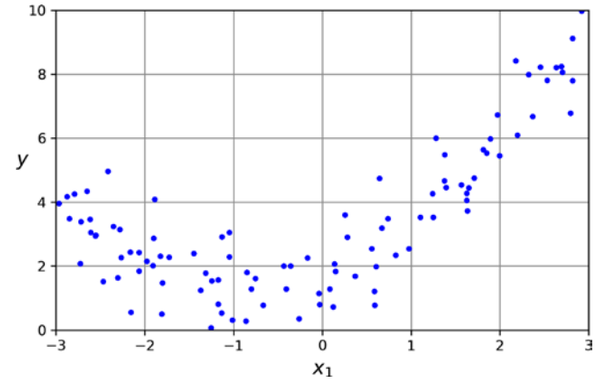



Figura 4.12. Conjunto de datos no lineales y con ruido generado.

Está claro que una línea recta nunca se ajustará a estos datos de forma adecuada. Así pues, vamos a utilizar la clase `PolynomialFeatures` de Scikit-Learn para transformar nuestros datos de entrenamiento, añadiendo el cuadrado (polinomio de segundo grado) de cada característica en el conjunto de entrenamiento como una característica nueva (en este caso, solo hay una característica):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

Ahora, `X_poly` contiene la característica original de `X` más el cuadrado de esta característica. Ya podemos ajustar un modelo `LinearRegression` a este conjunto de datos ampliado (véase la figura 4.13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

No está mal: el modelo estima $\hat{y} = 0{,}56x_1^2 + 0{,}93x_1 + 1{,}78$ cuando, en realidad, la función original era $y = 0{,}5x_1^2 + 1{,}0x_1 + 2{,}0 +$ ruido gaussiano.

Fíjate en que, cuando hay múltiples características, la regresión polinomial es capaz de encontrar relaciones entre características, que es algo que un modelo de regresión lineal simple no puede hacer. Esto es posible gracias al hecho de que `PolynomialFeatures` también añade todas las combinaciones de características hasta un grado determinado. Por ejemplo, si hubiese dos características *a* y *b*, `PolynomialFeatures` con `degree=3` no solo añadiría las características a^2, a^3, b^2 y b^3, sino también las combinaciones ab, a^2b y ab^2.

```
rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
                                   SVC(kernel="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

Este modelo se representa en la parte inferior izquierda de la figura 5.9. Los otros gráficos muestran modelos entrenados con diferentes valores de hiperparámetros `gamma` (γ) y `C`. Aumentar `gamma` hace que la curva con forma de campana se estreche (fíjate en los gráficos de la izquierda en la figura 5.8). Como resultado, el rango de influencia de cada instancia es más pequeño: el límite de decisión acaba siendo más irregular, serpenteando alrededor de instancias individuales. A la inversa, un valor `gamma` pequeño hace que la curva con forma de campana sea más ancha: las instancias tienen un rango de influencia mayor y el límite de decisión acaba siendo más suave. Por tanto, γ actúa como un hiperparámetro de regularización: si el modelo está sobreajustando, deberías reducir γ; si está subajustando, deberías incrementar γ (parecido al hiperparámetro `C`).

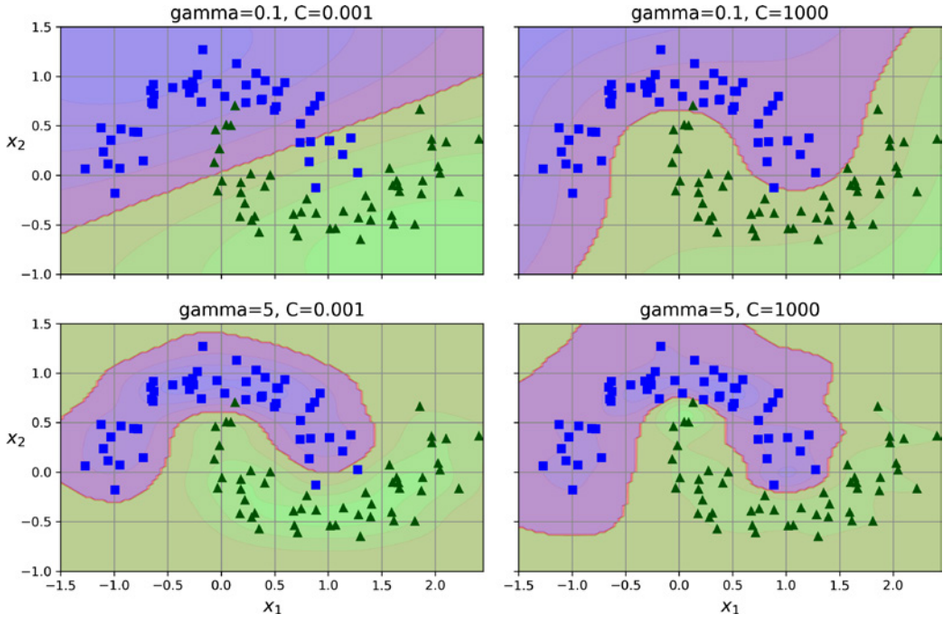


Figura 5.9. Clasificador SVM utilizando un *kernel* de función de base radial.

Existen otros *kernels*, pero se utilizan con mucha menos frecuencia. Algunos *kernels* son especializados para estructuras de datos específicas. A veces, se utilizan *kernels* de cadena cuando se clasifican documentos de texto o secuencias de ADN (por ejemplo, utilizando el *kernel* o los *kernels* de subsecuencia de cadena basados en la distancia de Levenshtein).

> **Truco:** Con tantos *kernels* entre los que elegir, ¿cómo decidir cuál utilizar? Como regla general, siempre deberías probar primero con el *kernel* lineal. La clase `LinearSVC` es mucho más rápida que `SVC(kernel="linear")`, sobre todo si el conjunto de entrenamiento es muy grande. Si no es demasiado grande, también deberías probar SVM kernelizadas, empezando por el *kernel* de función de base radial gaussiana; a menudo, funciona muy bien. Después, si tienes tiempo y potencia computacional de sobra, puedes experimentar con algunos *kernels* más, usando la búsqueda de hiperparámetros. Si hay *kernels* especializados para la estructura de datos de tu conjunto de entrenamiento, asegúrate de probarlos también.

Clases SVM y complejidad computacional

La clase `LinearSVC` se basa en la biblioteca `liblinear`, que implementa un algoritmo optimizado (`https://homl.info/13`) para SVM lineales.[1] No es compatible con el truco *kernel*, pero escala de manera casi lineal con el número de instancias de entrenamiento y el número de características. Su complejidad de tiempo de entrenamiento es aproximadamente $O(m \times n)$. El algoritmo tarda más si requieres una precisión muy alta. Esto está controlado por el hiperparámetro de tolerancia ϵ (llamado `tol` en Scikit-Learn). En la mayoría de las tareas de clasificación, la tolerancia predeterminada está bien.

La clase `SVC` se basa en la biblioteca `libsvm`, que implementa un algoritmo compatible con el truco *kernel* (`https://homl.info/14`).[2] La complejidad de tiempo de entrenamiento suele estar entre $O(m^2 \times n)$ y $O(m^3 \times n)$. Por desgracia, eso significa que va muy despacio cuando el número de instancias de entrenamiento es grande (por ejemplo, cientos de miles de instancias), así que este algoritmo es mejor para conjuntos de entrenamiento no lineales de tamaño pequeño o mediano. Escala bien con el número de características, sobre todo con características dispersas (es decir, cuando cada instancia tiene características con valores distintos de cero). En este caso, el algoritmo escala, aproximadamente, con el número medio de características distintas de cero por instancia.

La clase `SGDClassifier` también realiza una clasificación de margen amplio por defecto y sus hiperparámetros, sobre todo los hiperparámetros de regularización (`alpha` y `penalty`) y `learning_rate`, pueden ajustarse para producir resultados similares a las SVM lineales. Para el entrenamiento, utiliza el descenso de gradiente estocástico (véase el capítulo 4), que permite el aprendizaje gradual y usa poca memoria, así que puedes emplearlo para entrenar un modelo en un conjunto de datos grande que no quepa en la RAM (es decir, aprendizaje *out of core*). Además, escala muy bien, ya que su complejidad computacional es $O(m \times n)$. La tabla 5.1 compara clases de clasificación SVM de Scikit-Learn.

1. Chih-Jen Lin *et al.*, "A Dual Coordinate Descent Method for Large-Scale Linear SVM", actas de la 25.ª *International Conference on Machine Learning* (2008): 408-415.
2. John Platt, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines" (informe técnico de Microsoft Research, 21 de abril, 1998).

Límites de k-medias

Pese a todas sus virtudes, sobre todo la rapidez y la escalabilidad, k-medias no es perfecto. Como ya hemos visto, es necesario ejecutar el algoritmo varias veces para evitar soluciones no óptimas y también hay que especificar el número de grupos, lo cual puede ser bastante engorroso. Además, k-medias no funciona muy bien cuando los grupos tienen tamaños variados, densidades diferentes o formas no esféricas. Por ejemplo, la figura 9.11 muestra cómo k-medias agrupa un conjunto de datos que contiene tres grupos elipsoidales de diferentes dimensiones, densidades y orientaciones.

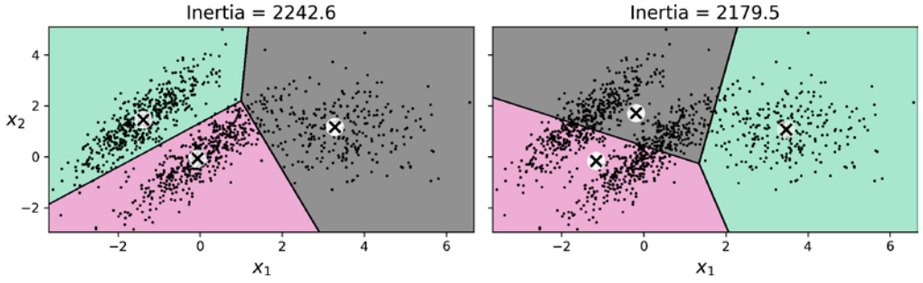


Figura 9.11. K-medias no consigue agrupar bien estas cinco masas elipsoidales.

Como ves, ninguna de estas soluciones es buena. La de la izquierda es mejor, pero, aun así, corta el 25 % del grupo central y lo asigna al grupo de la derecha. La solución de la derecha es terrible, a pesar de que su inercia es más baja. Por tanto, dependiendo de los datos, hay algoritmos de agrupamiento diferentes que pueden tener mejor rendimiento. En este tipo de grupos elípticos, funcionan genial los modelos de mezcla gaussiana.

> **Truco:** Es importante escalar las características de entrada (véase el capítulo 2) antes de ejecutar k-medias o los grupos podrían estirarse mucho y k-medias tendría un mal rendimiento. Escalar las características no garantiza que todos los grupos sean bonitos y esféricos, pero, por lo general, ayuda a k-medias.

Ahora vamos a echar un vistazo a algunas maneras en las que podemos beneficiarnos del agrupamiento. Utilizaremos k-medias, pero puedes experimentar a tu gusto con otros algoritmos de agrupamiento.

Utilizar agrupamiento para la segmentación de imágenes

La segmentación de imágenes es la tarea de dividir una imagen en múltiples segmentos. Hay diferentes variantes:

- En la segmentación por colores, los píxeles con un color similar se asignan al mismo segmento. Esto es suficiente para muchas aplicaciones. Por ejemplo, si queremos analizar imágenes por satélite para medir cuál es el área total de bosque en una región, la segmentación por colores podría bastar.
- En la segmentación semántica, todos los píxeles que son parte del mismo tipo de objeto se asignan al mismo segmento. Por ejemplo, en un sistema de visión para un coche con conducción autónoma, todos los píxeles que son parte de la imagen de un peatón podrían asignarse al segmento "peatón" (habría un segmento que contendría todos los peatones).
- En la segmentación de instancias, todos los píxeles que son parte del mismo objeto individual se asignan al mismo segmento. En este caso, habría un segmento diferente para cada peatón.

Lo más avanzado hoy en día respecto a la segmentación semántica o la de instancias se consigue utilizando arquitecturas complejas basadas en redes neuronales convolucionales (consulta el capítulo 14). En este capítulo, vamos a centrarnos en la tarea (mucho más simple) de la segmentación por colores, utilizando k-medias.

Vamos a empezar por importar el paquete Pillow (sucesor de la biblioteca Python Imaging Library, PIL), que, después, utilizaremos para cargar la imagen `ladybug.png` (véase la imagen superior izquierda de la figura 9.12), suponiendo que se encuentra en `filepath`:

```
>>> import PIL
>>> image = np.asarray(PIL.Image.open(filepath))
>>> image.shape
(533, 800, 3)
```

Figura 9.12. Segmentación de una imagen utilizando k-medias con varios números de grupos de colores.

La imagen se representa como una matriz 3D. El tamaño de la primera dimensión es la altura, el segundo es la anchura y el tercero es el número de canales de color, en este caso, rojo, verde y azul (RGB). Dicho de otro modo, cada píxel que hay ahí es un vector 3D que contiene

Aumento de datos

El aumento de datos incrementa de manera artificial el tamaño del conjunto de entrenamiento al generar muchas variantes realistas de cada instancia de entrenamiento. Eso reduce el sobreajuste, lo que lo convierte en una técnica de regularización. Las instancias generadas deberían ser tan realistas como sea posible: lo ideal sería que, al enseñar a un humano una imagen del conjunto de entrenamiento aumentado, no pudiese distinguir si estaba aumentado o no. Añadir ruido blanco, sin más, no servirá: las modificaciones deberían poder aprenderse (el ruido blanco no puede).

Por ejemplo, puedes desplazar, rotar y redimensionar cada imagen del conjunto de entrenamiento varias veces y añadir las imágenes resultantes a ese conjunto (véase la figura 14.13). Para ello, puedes utilizar capas de aumento de datos de Keras, introducidas en el capítulo 13 (por ejemplo, `RandomCrop`, `RandomRotation`, etc.). Eso obliga al modelo a ser más tolerante respecto a las variaciones en la posición, la orientación y el tamaño de los objetos en las imágenes. Para producir un modelo que sea más tolerante respecto a las condiciones de iluminación, puedes generar de manera similar muchas imágenes con varios contrastes. En general, también puedes poner las imágenes en horizontal (salvo el texto y otros objetos asimétricos). Al combinar estas transformaciones, puedes aumentar mucho el tamaño del conjunto de entrenamiento.

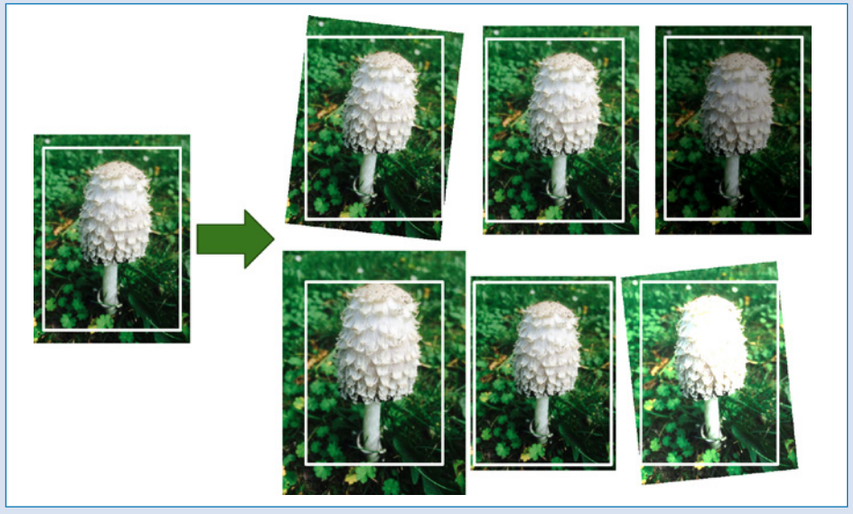


Figura 14.13. Generación de nuevas instancias de entrenamiento a partir de otras existentes.

El aumento de datos también resulta útil cuando tenemos un conjunto de datos desequilibrado: podemos utilizarlo para generar más muestras de las clases menos frecuentes. Esto se denomina SMOTE (*synthetic minority oversampling technique*, técnica de sobremuestreo de minorías sintéticas).

Para reducir el sobreajuste, los autores utilizaron dos técnicas de regularización. En primer lugar, aplicaron *dropout* (introducido en el capítulo 11) con una tasa de *dropout* del 50 % durante el entrenamiento a las salidas de las capas F9 y F10. En segundo lugar, llevaron a cabo un aumento de datos desplazando de manera aleatoria las imágenes de entrenamiento con distintas desviaciones, poniéndolas en horizontal y cambiando las condiciones de iluminación.

AlexNet utiliza también un paso de normalización competitiva inmediatamente después del paso ReLU de las capas C1 y C3, denominado "normalización de respuesta local" (LRN, *local response normalization*): las neuronas activadas con más fuerza inhiben otras neuronas situadas en la misma posición en mapas de características vecinos. Esta activación competitiva se ha observado en las neuronas biológicas. Esto impulsa a diferentes mapas de características a especializarse, separándolos y obligándolos a explorar un rango más amplio de características, mejorando, finalmente, la generalización. La ecuación 14.2 muestra cómo aplicar la LRN.

Ecuación 14.2. Normalización de respuesta local (LRN).

$$b_i = a_i\left(k + \alpha \sum_{j=j_{\text{bajo}}}^{j_{\text{alto}}} a_j^{\,2}\right)^{-\beta} \quad \text{con} \quad \begin{cases} j_{\text{alto}} = \text{mín.}\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{bajo}} = \text{máx.}\left(0, i - \frac{r}{2}\right) \end{cases}$$

En esta ecuación:

- b_i es la salida normalizada de la neurona ubicada en el mapa de características i, en una fila u y columna v (ten en cuenta que en esta ecuación consideramos solo neuronas situadas en esa fila y columna, así que u y v no se muestran).
- a_i es la activación de esa neurona después del paso ReLU, pero antes de la normalización.
- k, α, β y r son hiperparámetros. k es el sesgo y r es el radio de profundidad.
- f_n es el número de mapas de características.

Por ejemplo, si $r = 2$ y una neurona tiene una activación fuerte, inhibirá la activación de las neuronas situadas en los mapas de características inmediatamente encima y debajo del suyo.

En AlexNet, los hiperparámetros se configuran de la siguiente manera: $r = 5$, $\alpha = 0{,}0001$, $\beta = 0{,}75$ y $k = 2$. Este paso puede implementarse utilizando la función `tf.nn.local_response_normalization()` (que puedes envolver en una capa `Lambda` si quieres utilizarla en un modelo Keras).

Matthew Zeiler y Rob Fergus desarrollaron una variante de AlexNet llamada ZF Net (`https://homl.info/zfnet`),[12] que ganó el reto ILSVRC en 2013. En esencia, es AlexNet con varios hiperparámetros ajustados (número de mapas de características, tamaño del *kernel*, paso de avance, etc.).

12. Matthew D. Zeiler y Rob Fergus, "Visualizing and Understanding Convolutional Networks", actas de la *European Conference on Computer Vision* (2014): 818-833.

```
dropout_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
dropout_ae = tf.keras.Sequential([dropout_encoder, dropout_decoder])
```

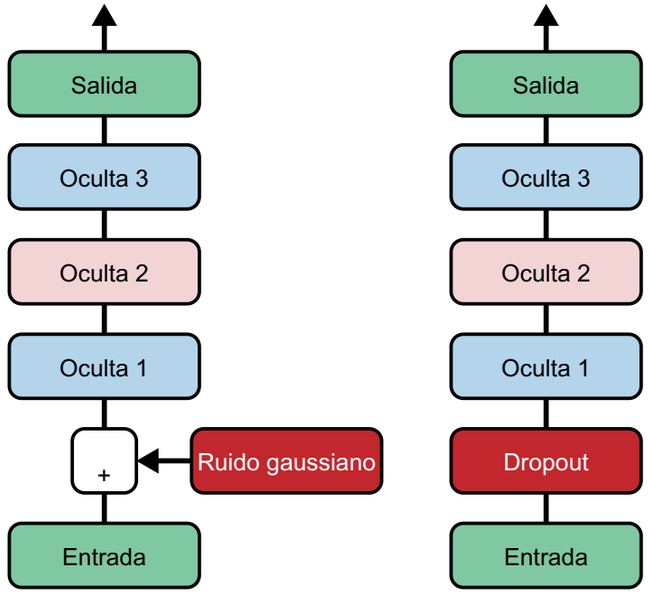


Figura 17.8. Autocodificadores con eliminación de ruido, con ruido gaussiano (izquierda) o *dropout* (derecha).

La figura 17.9 muestra algunas imágenes con ruido (con la mitad de los píxeles desactivados) y las imágenes reconstruidas por el autocodificador con eliminación de ruido basado en *dropout*. Fíjate en cómo el autocodificador supone detalles que en realidad no están en la entrada, como la parte superior de la camisa blanca (fila inferior, cuarta imagen). Como ves, los autocodificadores con eliminación de ruido no solo pueden utilizarse para la visualización de datos o el preentrenamiento no supervisado, como los otros autocodificadores de los que hemos hablado hasta ahora, sino que también pueden utilizarse de forma bastante simple y eficiente para eliminar ruido de imágenes.

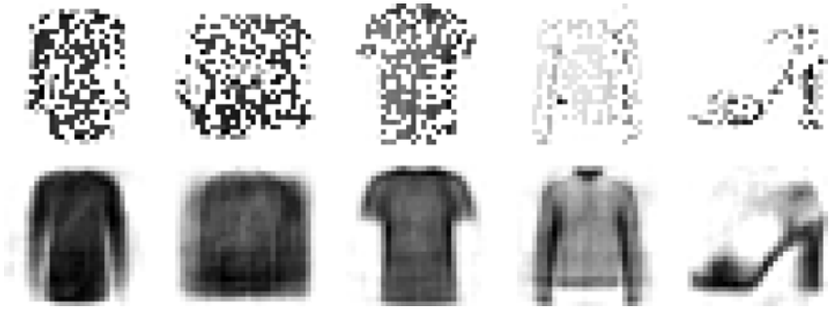


Figura 17.9. Imágenes con ruido (arriba) y sus reconstrucciones (abajo).

Autocodificadores dispersos

Otro tipo de limitación que suele conducir a una buena extracción de características es la dispersión: al añadir un término apropiado a la función de pérdida, el autocodificador se ve obligado a reducir el número de neuronas activas en la capa de codificación. Por ejemplo, podría verse obligado a tener de media solo 5 % neuronas significativamente activas en la capa de codificación. Esto obliga al autocodificador a representar cada entrada como una combinación de un número pequeño de activaciones. Como resultado, cada neurona de la capa de codificación suele acabar representando una característica útil (si solo pudieses decir unas pocas palabras al mes, lo más probable es que intentases que mereciese la pena escucharlas).

Un enfoque simple es utilizar la función de activación sigmoide en la capa de codificación (para restringir las codificaciones a valores entre 0 y 1), usar una capa de codificación grande (por ejemplo, con 300 unidades) y añadir cierta regularización ℓ_1 a las activaciones de la capa de codificación.

El descodificador es solo un descodificador corriente:

```
sparse_l1_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid"),
    tf.keras.layers.ActivityRegularization(l1=1e-4)
])
sparse_l1_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
sparse_l1_ae = tf.keras.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

El proceso de autenticación se basa en OAuth 2.0 (`https://oauth.net`): una ventana emergente te pedirá que confirmes que quieres que el cuaderno de Colab acceda a tus credenciales de Google. Si aceptas, debes seleccionar la misma cuenta de Google que hayas usado para GCP. Después, se te pedirá que confirmes que estás de acuerdo con dar a Colab acceso total a todos tus datos en Google Drive y en GCP. Si lo permites, solo tendrá acceso el cuaderno actual y solo hasta que expire el entorno de ejecución de Colab. Obviamente, solo deberías aceptar esto si confías en el código del cuaderno.

> **Advertencia:** Si no estás trabajando con los cuadernos oficiales de `https://github.com/ageron/handson-ml3`, deberías tener mucho cuidado; si el autor del cuaderno es malvado, podría incluir código para hacer lo que quiera con tus datos.

Autenticación y autorización en GCP

En general, el uso de la autenticación OAuth 2.0 solo se recomienda cuando una aplicación debe acceder a los datos personales o recursos del usuario desde otra aplicación, en nombre del usuario. Por ejemplo, algunas aplicaciones permiten al usuario guardar datos en su Google Drive, pero, para ello, primero la aplicación necesita que el usuario se autentique con Google y permita el acceso a Google Drive. En general, la aplicación solo pedirá el nivel de acceso que necesita; no será un acceso ilimitado: por ejemplo, la aplicación solo solicitará acceso a Google Drive, no a Gmail o a cualquier otro servicio de Google. Además, la autorización suele expirar después de un tiempo, y siempre puede revocarse. Cuando una aplicación necesita acceso a un servicio de GCP en su propio nombre, no en nombre del usuario, normalmente debería utilizar una cuenta de servicio. Por ejemplo, si creas un sitio web que necesita enviar peticiones de predicciones a un punto final Vertex AI, el sitio web estará accediendo al servicio en su propio nombre. No hay datos ni recursos a los que necesite acceder en la cuenta de Google del usuario. De hecho, muchos usuarios del sitio web ni siquiera tendrán una cuenta de Google. Para este escenario, primero necesitas crear una cuenta de servicio. Selecciona **IAM y administración>Cuentas de servicio** en el menú de navegación ☰ de la consola GCP (o utiliza el cuadro de búsqueda), después haz clic en **+ Crear cuenta de servicio**, rellena la primera página del formulario (nombre de la cuenta del servicio, ID, descripción) y haz clic en **Crear y continuar**. A continuación, debes dar a esta cuenta algunos derechos de acceso. Selecciona el rol **Usuario de Vertex AI**: esto permitirá a la cuenta de servicio realizar predicciones y utilizar otros servicios de Vertex AI, pero nada más. Haz clic en **Continuar**. Ahora puedes, de manera opcional, dar a algunos usuarios acceso a la cuenta de servicio: esto resulta útil cuando tu cuenta de usuario de GCP es parte de una organización y quieres autorizar a otros usuarios de la organización a que desplieguen aplicaciones que se basarán en esta cuenta de servicio o a gestionar la cuenta de servicio en sí. A continuación, haz clic en **Listo**. Una vez que hayas creado una cuenta de servicio, tu aplicación debe autenticarse como esa cuenta de servicio. Hay varias formas de hacerlo. Si la aplicación se aloja en GCP (por ejemplo, si estás escribiendo código para un sitio web que se aloja en Google Compute Engine), la solución más simple y segura es adjuntar la cuenta de servicio al recurso de GCP que aloja tu sitio web, como una instancia de máquina virtual o un servicio de Google App Engine. Esto puede hacerse al crear el recurso de GCP, seleccionando la cuenta de servicio en la sección **Identidad y acceso de API**. Algunos recursos, como las instancias de máquina virtual, también te permiten adjuntar la cuenta de servicio después de haber creado la instancia de máquina virtual: debes detenerla y editar su configuración. En cualquier caso, una vez que una cuenta de servicio se adjunta a una instancia de máquina virtual o cualquier otro recurso de GCP que ejecute tu código, las bibliotecas de clientes de GCP (que veremos enseguida) la autenticarán de manera automática como la cuenta de servicio elegida, sin que se necesiten más pasos.

Si tu aplicación se aloja utilizando Kubernetes, deberías utilizar el servicio Workload Identity de Google para asignar la cuenta de servicio adecuada a cada cuenta de servicio Kubernetes.

Si tu aplicación no se aloja en GCP (por ejemplo, si estás ejecutando un cuaderno de Jupyter en tu propia máquina), puedes utilizar el servicio **Federación de identidades para cargas de trabajo** (que es la manera más segura, pero más difícil) o generar sin más una clave de acceso para tu cuenta de servicio, guardarla en un archivo JSON, y apuntar la variable de entorno `GOOGLE_APPLICATION_CREDENTIALS` hacia ella para que la aplicación cliente pueda acceder a ella. Puedes gestionar las claves de acceso haciendo clic en la cuenta de servicio que acabas de crear y, después, abriendo la pestaña **Claves**. Asegúrate de mantener el archivo de clave secreto: es como una contraseña para la cuenta de servicio.

Para obtener más detalles sobre la configuración de la autenticación y la autorización para que tu aplicación pueda acceder a servicios de GCP, consulta la documentación (`https://homl.info/gcpauth`).

Ahora, vamos a crear un *bucket* (depósito) en Google Cloud Storage para almacenar nuestro SavedModels (un *bucket* de GCS es un contenedor para nuestros datos). Para ello, vamos a utilizar la biblioteca `google-cloud-storage`, que está preinstalada en Colab. Primero, creamos un objeto `Client`, que servirá como interfaz con GCS y, después, lo utilizamos para crear el *bucket*:

```
from google.cloud import storage

project_id = "my_project"  # cambia esto por el ID de tu proyecto
bucket_name = "my_bucket"  # cambia esto por un nombre de bucket único
location = "us-central1"

storage_client = storage.Client(project=project_id)
bucket = storage_client.create_bucket(bucket_name, location=location)
```

> **Truco:** Si quieres reutilizar un *bucket* existente, sustituye la última línea con `bucket = storage_client.bucket(bucket_name)`. Asegúrate de que la ubicación está configurada como la región del *bucket*.

Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow

Gracias a varios logros innovadores, el *deep learning* ha dado un gran impulso a todo el campo del *machine learning*. Ahora, incluso programadores que no saben casi nada de esta tecnología pueden usar herramientas sencillas y eficaces para implementar programas capaces de aprender a partir de datos. Este *best seller* utiliza ejemplos concretos, una teoría mínima y *frameworks* de Python listos para la producción (Scikit-Learn, Keras y TensorFlow) para ayudarte a obtener una comprensión intuitiva de los conceptos y herramientas para crear sistemas inteligentes.

Con esta tercera edición actualizada, el autor Aurélien Géron explora una variedad de técnicas que van desde una regresión lineal simple a redes neuronales profundas. Hay ejemplos de código y ejercicios por todo el libro para ayudarte a aplicar lo que has aprendido, lo único que necesitas para empezar es experiencia en programación.

- Utiliza Scikit-Learn para hacer un seguimiento de un proyecto de *machine learning* de ejemplo de principio a fin.
- Explora varios modelos, incluyendo máquinas de vectores soporte, árboles de decisión, *random forests* y métodos de ensamblaje.
- Aprovecha técnicas de aprendizaje no supervisado, como la reducción de dimensionalidad, el agrupamiento y la detección de anomalías.
- Sumérgete en arquitecturas de redes neuronales, incluyendo redes convolucionales, redes recurrentes, redes generativas antagónicas, autocodificadores, modelos de difusión y transformadores.
- Utiliza TensorFlow y Keras para crear y entrenar redes neuronales para visión por ordenador, procesamiento del lenguaje natural, modelos generativos y aprendizaje profundo por refuerzo.

Aurélien Géron es asesor de *machine learning*. Antiguo *googler*, dirigió el equipo de clasificación de vídeos de YouTube desde 2013 a 2016. También fue fundador y CTO de Wifirst (proveedor de servicios de Internet inalámbrico líder en Francia) desde 2002 a 2012 y fundador y CTO de la empresa consultora de telecomunicaciones Polyconseil en 2001.

«Un recurso excepcional para estudiar *machine learning*. Encontrarás explicaciones claras e intuitivas y un montón de trucos prácticos».

—François Chollet
autor de Keras, autor de *Deep Learning con Python*

«Este libro es una gran introducción a la teoría y la práctica de la resolución de problemas con redes neuronales; se lo recomiendo a cualquiera que quiera aprender sobre *machine learning* práctico».

—Pete Warden
Mobile Lead de TensorFlow

ISBN 978-84-415-4804-6
2315192
9 788441 548046

ANAYA MULTIMEDIA
O'REILLY®
www.anayamultimedia.es