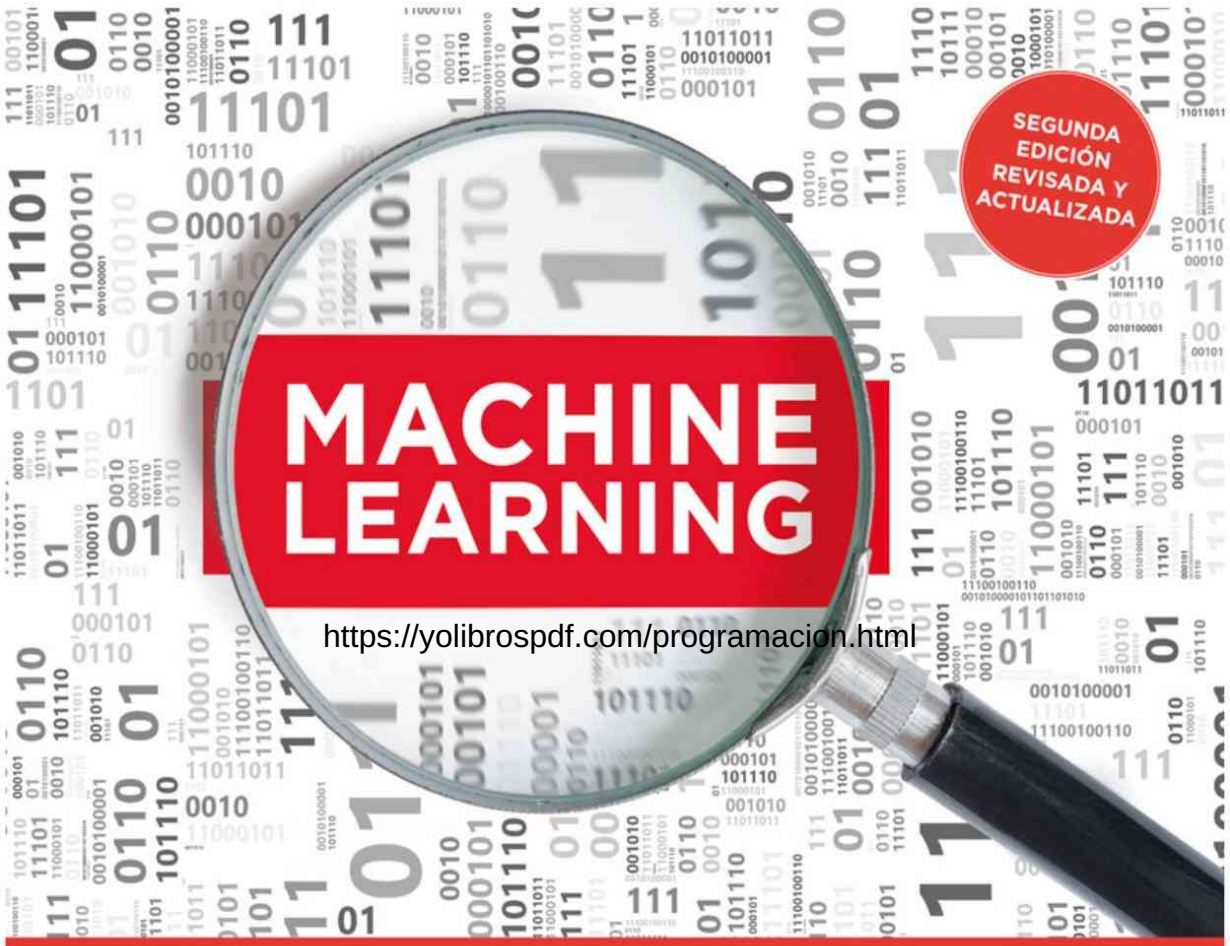


Sebastian Raschka · Vahid Mirjalili

Python Machine Learning

Aprendizaje automático y aprendizaje profundo con Python, scikit-learn y TensorFlow

Marcombo

Aprendizaje automático con Python

Aprendizaje automático y aprendizaje profundo
con Python, scikit-learn y TensorFlow

Segunda edición

Aprendizaje automático con Python

Aprendizaje automático y aprendizaje profundo con Python, scikit-learn y TensorFlow

Sebastian Raschka
Vahid Mirjalili

Segunda edición original publicada en inglés por Packt Publishing Ltd. con el título: *Python Machine Learning*, © 2017 Sebastian Raschka y Vahid Mirjalili
Título de la edición en español: *Aprendizaje automático con Python*
Segunda edición en español, año 2019

www.marcombo.com

Traducción: Sònia Llena
Revisor técnico: Ferran Fàbregas
Correctora: Anna Alberola
Directora de producción: M.ª Rosa Castillo

ISBN: 978-84-267-2720-6
D.L.: B-27539-2018
Impreso en Servicepoint
Printed in Spain

Sobre los autores

Sebastian Raschka, autor del libro líder de ventas *Python Machine Learning* [Aprendizaje automático con Python], cuenta con años de experiencia en codificación en Python, y ha impartido muchos seminarios sobre aplicaciones prácticas de ciencia de datos, aprendizaje automático y aprendizaje profundo. También es autor de un tutorial de aprendizaje automático en SciPy (el congreso líder en computación científica en Python).

Si bien los proyectos académicos de investigación de Sebastian se centran principalmente en la resolución de problemas en biología computacional, lo que a él le gusta es escribir y hablar sobre ciencia de datos, aprendizaje automático y Python en general; le motiva ayudar a la gente a que desarrolle soluciones basadas en datos sin tener necesariamente una base en aprendizaje automático.

Su trabajo y sus contribuciones han sido recientemente reconocidos con el premio *Departmental outstanding graduate student* 2016-2017, así como con el *ACM Computing Reviews' Best of 2016*. En su tiempo libre, a Sebastian le gusta colaborar en proyectos de código abierto, y los métodos que ha implementado se utilizan actualmente con éxito en concursos sobre aprendizaje automatico, como Kaggle.

> Me gustaría aprovechar esta oportunidad para dar las gracias a la excelente comunidad de Python y a los desarrolladores de paquetes de código abierto que me han ayudado a crear el entorno perfecto para la investigación científica y la ciencia de datos. También quiero dar las gracias a mis padres, quienes siempre me han animado y me han apoyado para seguir el camino y la carrera que tanto me apasionaba.
>
> Quiero dar las gracias especialmente a los principales desarrolladores de scikit-learn. Como colaborador en su proyecto, he tenido el placer de

trabajar con personas excelentes que no solo tienen grandes conocimientos sobre el aprendizaje automático sino que también son grandes programadores. Por último, me gustaría dar las gracias a Elie Kawerk, quien ha revisado de forma voluntaria este libro y me ha proporcionado un valioso *feedback* par los nuevos capítulos.

Vahid Mirjalili obtuvo su doctorado en ingeniería mecánica trabajando en métodos innovadores para simulaciones computacionales a gran escala de estructuras moleculares. Actualmente, centra su trabajo de investigación en aplicaciones de aprendizaje automático en distintos proyectos de visión por ordenador en el departamento de ciencia computacional e ingeniería de la Michigan State University.

Vahid eligió Python como su lenguaje de programación número uno, y durante su carrera de investigación y académica ha adquirido una gran experiencia en la codificación en Python. Aprendió a programar con Python en las clases de ingeniería en la Michigan State University, que le dieron la oportunidad de ayudar a otros estudiantes a entender distintas estructuras de datos y a desarrollar eficazmente código en Python.

Si bien gran parte de los intereses de investigación de Vahid se centran en las aplicaciones de aprendizaje profundo y de visión por ordenador, siente interés especialmente por equilibrar técnicas de aprendizaje profundo para ampliar la privacidad en datos biométricos, como las imágenes del rostro para que la información no se revele más de lo que los usuarios intentan revelar. Además, también colabora con un equipo de ingenieros que trabajan en coches autónomos, donde él diseña modelos de redes neuronales para la fusión de imágenes multiespectrales para la detección de peatones.

Me gustaría dar las gracias a mi mentor de doctorado, Dr. Arun Ross, por darme la oportunidad de trabajar en nuevos problemas en su laboratorio de investigación. También quiero dar las gracias al Dr. Vishnu Boddeti por despertar mi interés en el aprendizaje profundo y desmitificar sus conceptos básicos.

Introducción

Debido a su aparición en noticias y redes sociales, probablemente eres consciente de que el *machine learning* o aprendizaje automático se ha convertido en una de las tecnologías más apasionantes de nuestros tiempos. Grandes compañías, como Google, Facebook, Apple, Amazon e IBM, han invertido fuertemente en aplicaciones e investigación de aprendizaje automático por buenas razones. Si bien puede parecer que el aprendizaje automático se ha convertido en la palabra de moda de nuestros tiempos, la verdad es que no se trata de una moda pasajera. Este apasionante campo abre la puerta a nuevas posibilidades y se ha convertido en indispensable en nuestra vida diaria. Y esto se evidencia cuando hablamos con el asistente de voz en nuestros teléfonos inteligentes, recomendamos el producto adecuado a nuestros clientes, evitamos fraudes con las tarjetas de crédito, filtramos el correo no deseado para que no entre en nuestra bandeja de entrada o detectamos y diagnosticamos enfermedades. Y la lista no acaba aquí.

Si te quieres dedicar al aprendizaje automático, si quieres mejorar la resolución de problemas, o si quizás estás considerando hacer una carrera en investigación sobre aprendizaje automático, este libro es para ti. Sin embargo, para un principiante, los conceptos teóricos que se esconden detrás del aprendizaje automático pueden ser bastante abrumadores. En los últimos años, se han publicado muchos libros prácticos que pueden ayudarte a empezar con el aprendizaje automático mediante la implementación de potentes algoritmos de aprendizaje.

El planteamiento de ejemplos de código prácticos y el trabajo con aplicaciones de ejemplo de aprendizaje automático son una excelente forma de profundizar en este campo. Los ejemplos concretos ayudan a ilustrar los amplios conceptos poniendo en práctica directamente el material aprendido. ¡Pero recuerda que un gran poder conlleva una gran responsabilidad! Además de

proporcionar una experiencia práctica con el aprendizaje automático mediante los lenguajes de programación de Python y las librerías de aprendizaje automático basadas en Python, este libro presenta los conceptos matemáticos que se esconden detrás de los algoritmos del aprendizaje automático, aspecto esencial para que el uso del aprendizaje automático sea un éxito. Por lo tanto, este libro no es estrictamente un libro práctico; es un libro que trata los detalles necesarios relacionados con los conceptos del aprendizaje automático y ofrece explicaciones intuitivas y, al mismo tiempo, informativas acerca de cómo trabajan los algoritmos del aprendizaje automático, cómo utilizarlos y, lo más importante, cómo evitar los errores más comunes.

Actualmente, si escribes «aprendizaje automático» en Google Académico obtienes un abrumador número de resultados: más de 90 000. Evidentemente, no podemos discutir la esencia de todos los diferentes algoritmos y aplicaciones que han surgido en los últimos 60 años. Sin embargo, en este libro emprenderemos un apasionante viaje que recorre todos los conceptos y temas esenciales para que tengas un buen comienzo en este campo. Si crees que tu sed de conocimientos no queda satisfecha, este libro contiene referencias a múltiples recursos útiles que pueden servir para seguir adelante con los avances esenciales en este campo.

Si ya has estudiado antes la teoría del aprendizaje automático en detalle, este libro te mostrará cómo poner en práctica todo cuanto sabes. Si has utilizado antes técnicas de aprendizaje automático y deseas obtener más información acerca de cómo funciona realmente el aprendizaje automático, este libro es para ti. Y no te preocupes si eres completamente nuevo en este campo; todavía tienes más razones para estar emocionado. Te prometemos que el aprendizaje automático cambiará la manera que tienes de pensar en los problemas que quieres resolver y te mostrará cómo abordarlos desbloqueando el poder de los datos.

Antes de ir más lejos en el campo del aprendizaje automático, vamos a dar respuesta a tu pregunta más importante: «¿Por qué Python?». La respuesta es simple: porque es potente y, a la vez, muy accesible. Python se ha convertido en el lenguaje de programación más popular para la ciencia de datos porque permite que nos olvidemos de la parte tediosa de la programación y nos ofrece un entorno donde podemos anotar rápidamente nuestras ideas y poner los conceptos directamente en acción.

Nosotros, los autores, podemos decir de verdad que el estudio del aprendizaje automático nos ha hecho mejores científicos, mejores pensadores y mejores solucionadores de problemas. En este libro, queremos compartir contigo estos conocimientos. El conocimiento se obtiene aprendiendo. La clave se encuentra en nuestro entusiasmo y el verdadero dominio de las habilidades solo se puede lograr con la práctica. El camino a recorrer puede estar, en ocasiones, lleno de baches y algunos temas pueden ser más desafiantes que otros, pero esperamos que aceptes esta oportunidad y te centres en la recompensa. Recuerda que estamos juntos en este viaje y que, con este libro, vamos a añadir poderosas técnicas a tu arsenal que nos ayudarán a resolver incluso los problemas más difíciles planteados por datos.

Qué contiene este libro

El *Capítulo 1, Dar a los ordenadores el poder de aprender de los datos* presenta las principales subáreas del aprendizaje automático para resolver distintas tareas problemáticas. Además, trata sobre los pasos esenciales para crear un modelo típico de aprendizaje automático mediante la construcción de un entramado que nos guiará a través de los siguientes capítulos.

El *Capítulo 2, Entrenar algoritmos simples de aprendizaje automático para clasificación* vuelve a los orígenes del aprendizaje automático y presenta clasificadores binarios basados en perceptrones y neuronas lineales adaptativas. Este capítulo es una breve introducción a los fundamentos de la clasificación de patrones y se centra en la interacción de algoritmos de optimización y aprendizaje automático.

El *Capítulo 3, Un recorrido por los clasificadores de aprendizaje automático con scikit-learn* describe los algoritmos básicos del aprendizaje automático para clasificar y proporciona ejemplos prácticos con una de las librerías de aprendizaje automático de código abierto más exhaustiva y conocida: scikit-learn.

El *Capítulo 4, Generar buenos modelos de entrenamiento: preprocesamiento de datos* trata sobre cómo enfrentarnos a los problemas más comunes de conjuntos de datos no procesados, como los datos incompletos. También trata varios enfoques para identificar las características más informativas en los conjuntos de datos y muestra cómo preparar variables de diferentes tipos, como entradas correctas para algoritmos de aprendizaje automático.

El *Capítulo 5, Comprimir datos mediante la reducción de dimensionalidad* describe las técnicas esenciales para reducir el número de características en un conjunto de datos a conjuntos más pequeños manteniendo la mayor parte de su información útil y discriminatoria. Trata del enfoque estándar de la reducción de dimensionalidad mediante el análisis del componente principal y lo compara con técnicas de transformación no lineales y supervisadas.

El *Capítulo 6, Aprender las mejores prácticas para la evaluación de modelos y el ajuste de hiperparámetros* habla de lo que se debe y no se debe hacer para estimar los resultados de los modelos predictivos. Además, analiza distintos parámetros para medir los resultados de nuestros modelos y técnicas para ajustar con precisión los algoritmos del aprendizaje automático.

El *Capítulo 7, Combinar diferentes modelos para el aprendizaje conjunto* presenta los diferentes conceptos para combinar de manera efectiva múltiples algoritmos de aprendizaje. Muestra cómo crear sistemas expertos para superar las debilidades de aprendizajes individuales, que den como resultado unas predicciones más precisas y fiables.

El *Capítulo 8, Aplicar el aprendizaje automático para el análisis de sentimiento* trata los pasos esenciales para transformar datos textuales en representaciones con significado para los algoritmos del aprendizaje automático, para así predecir las opiniones de la gente en base a su escritura.

El *Capítulo 9, Incrustar un modelo de aprendizaje automático en una aplicación web* retoma el modelo predictivo del capítulo anterior y continúa con los pasos esenciales del desarrollo de aplicaciones web con modelos de aprendizaje automático incrustados.

El *Capítulo 10, Predicción de variables de destino continuas con análisis de regresión* habla de las técnicas esenciales para relaciones lineales de modelado entre destino y variables de respuesta para hacer predicciones en una escala continua. Después de introducir diferentes modelos lineales, también trata de la regresión polinomial y los enfoques basados en árboles.

El *Capítulo 11, Trabajar con datos sin etiquetar: análisis de grupos* cambia el enfoque a una subárea distinta del aprendizaje automático: el aprendizaje no supervisado. Aplicamos algoritmos de tres familias básicas de algoritmos agrupados para encontrar grupos de objetos que compartan un cierto grado de semejanza.

El *Capítulo 12, Implementar una red neuronal artificial multicapa desde cero* amplía el concepto de optimización basada en gradiente, que introducimos por primera vez en el *Capítulo 2, Entrenando algoritmos simples del aprendizaje automático para la clasificación*, para construir potentes redes neuronales multicapas basadas en el popular algoritmo *backpropagation* de Python.

El *Capítulo 13, Paralelización de entrenamiento de redes neuronales con TensorFlow* se basa en cuanto se ha aprendido en el capítulo anterior para proporcionar una guía práctica para el entrenamiento más eficaz de redes neuronales. Este capítulo se centra en TensorFlow, una librería de Python de código abierto que nos permite utilizar múltiples núcleos de GPU modernas.

El *Capítulo 14, Ir más lejos: la mecánica de TensorFlow* cubre TensorFlow de un modo mucho más detallado, explicando sus conceptos básicos de sesiones y gráficos computacionales. Además, este capítulo trata temas como guardar y visualizar gráficos de redes neuronales, que serán muy útiles para el resto de capítulos del libro.

El *Capítulo 15, Clasificar imágenes con redes neuronales convolucionales profundas* habla de la arquitectura de las redes neuronales profundas, que se ha convertido en el nuevo estándar en campos de visión artificial y reconocimiento de imágenes (redes neuronales convolucionales). Este capítulo tratará los principales conceptos entre capas convolucionales, como extractor de características, y la aplicación de arquitecturas de redes neuronales convolucionales a una tarea de clasificación de imágenes para conseguir una precisión casi perfecta en la clasificación.

El *Capítulo 16, Modelado de datos secuenciales mediante redes neuronales recurrentes* presenta otra arquitectura de red neuronal popular para el aprendizaje profundo que es muy aconsejable, especialmente para trabajar con datos secuenciales y datos de series de tiempo. En este capítulo, aplicaremos diferentes arquitecturas de redes neuronales recurrentes a datos textuales.

Empezaremos con una tarea de análisis de sentimientos como ejercicio de calentamiento y aprenderemos a generar por completo un nuevo texto.

Qué necesitas para este libro

La ejecución de los ejemplos de código proporcionados en este libro requiere una instalación de Python 3.6.0 o posterior en macOS, Linux o Microsoft Windows. Con frecuencia utilizaremos librerías básicas de Python para computación científica, como SciPy, NumPy, scikit-learn, Matplotlib y pandas.

En el primer capítulo encontrarás instrucciones y consejos útiles para configurar tu entorno de Python y estas librerías básicas. Añadiremos librerías adicionales a nuestro repertorio. Además, en los correspondientes capítulos también se proporcionan instrucciones de instalación: la librería NLTK para procesamiento de lenguaje natural (*Capítulo 8, Aplicar el aprendizaje automático para el análisis de sentimientos*), el *framework* de desarrollo web Flask (*Capítulo 9, Incrustar un modelo de aprendizaje automático en una aplicación web*), la librería Seaborn para la visualización de datos estadísticos (*Capítulo 10, Predicción de variables de destino continuas con análisis de regresión*) y TensorFlow para el entrenamiento eficaz de redes neuronales en unidades de procesamiento gráfico (*Capítulos 13 a 16*).

A quién va dirigido este libro

Si te interesa saber cómo utilizar Python para empezar a dar respuesta a cuestiones importantes sobre tus datos, elige *Aprendizaje automático con Python - Segunda edición*. Tanto si deseas empezar desde cero como si deseas ampliar tus conocimientos científicos sobre datos, este es un recurso esencial e ineludible.

Convenciones

En este libro, encontrarás múltiples estilos de texto que distinguen entre diferentes tipos de información. A continuación, verás algunos ejemplos de estos estilos y una explicación de su significado.

Código en texto, nombres de tablas de bases de datos, nombres de directorios, nombres de archivos, extensiones de archivo, nombres de ruta, URL ficticias, entradas de usuario y controles de Twitter se muestran del siguiente modo: «Con la configuración **out_file=None**, asignamos directamente el dato de punto a una variable **dot_data**, en lugar de escribir un fichero **tree.dot** intermedio al disco».

Un bloque de código se indica de este modo:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
... metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
... classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.show()
```

Todas las líneas de comando de entrada y salida se escriben así:

```
pip3 install graphviz
```

Los **nuevos términos** y las **palabras importantes** se muestran en negrita. Las palabras que ves en pantalla, por ejemplo en menús o cuadros de diálogo, aparecen en el texto de este modo: «Después de hacer clic en el botón **Dashboard** de la esquina superior derecha, accedemos al panel de control que se muestra en la parte superior de la página».

Advertencias o notas importantes se

muestran en un cuadro como este.

Trucos y consejos se muestran así.

Descargar el código de ejemplo y las imágenes en color de este libro

En la parte inferior de la primera página del libro encontrarás el código de acceso que te permitirá descargar de forma gratuita los contenidos adicionales del libro.

Índice

1

Dar a los ordenadores el poder de aprender de los datos

En mi opinión, el **aprendizaje automático** –la aplicación y ciencia de los algoritmos que da sentido a los datos– es el campo más apasionante de todas las ciencias computacionales. Vivimos en una época en la cual los datos llegan en abundancia; utilizando algoritmos de autoaprendizaje del campo del aprendizaje automático podemos convertir estos datos en conocimiento. Gracias a las múltiples y potentes librerías de código abierto que han sido desarrolladas en los últimos años, probablemente no ha habido un momento mejor para acceder al campo del aprendizaje automático y aprender cómo utilizar potentes algoritmos para detectar patrones de datos y hacer predicciones sobre acontecimientos futuros.

En este capítulo, aprenderás los principales conceptos y los diferentes tipos de aprendizaje automático. Junto con una introducción básica a la terminología más importante, sentaremos las bases para utilizar con éxito técnicas de aprendizaje automático para la resolución práctica de problemas.

En este capítulo, trataremos los siguientes temas:

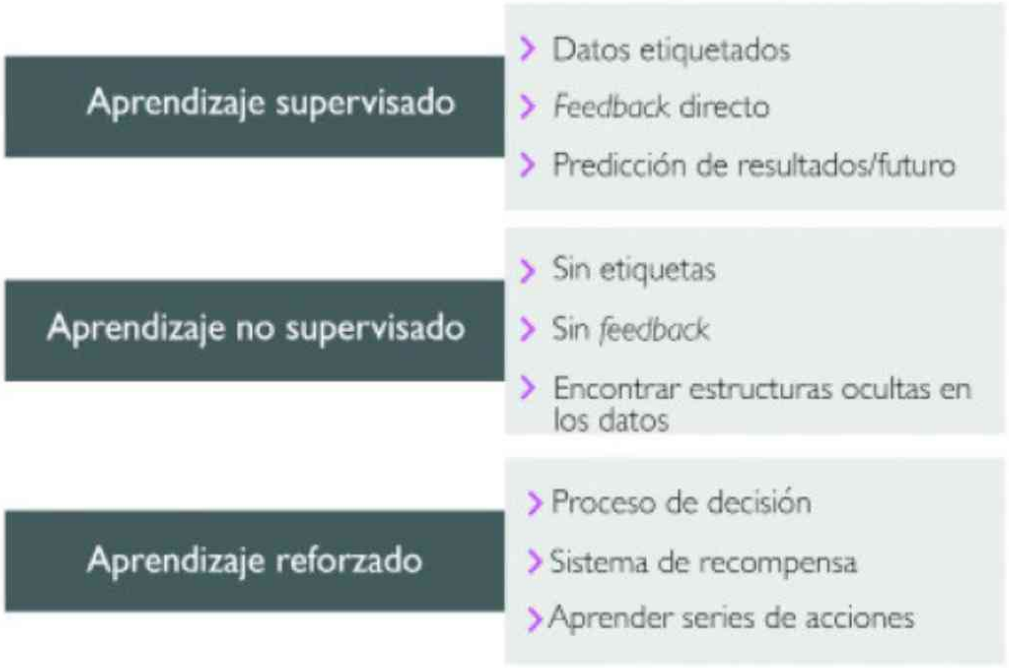
- Los conceptos generales del aprendizaje automático.
- Los tres tipos de aprendizaje y la terminología básica.
- La construcción de bloques para diseñar sistemas de aprendizaje automático.
- La instalación y configuración de Python para el análisis de datos y el aprendizaje automático.

Crear máquinas inteligentes para transformar datos en conocimiento

En esta época de tecnología moderna, existe un recurso que tenemos en abundancia: gran cantidad de datos estructurados y no estructurados. En la segunda mitad del siglo veinte, el aprendizaje automático evolucionó como un subcampo de la **Inteligencia Artificial (AI)** que involucraba algoritmos de autoaprendizaje que derivaban el conocimiento a partir de datos para crear predicciones. En lugar de necesitar al hombre para derivar de forma manual las reglas y crear modelos a partir del análisis de grandes cantidades de datos, el aprendizaje automático ofrece una alternativa más eficiente para capturar el conocimiento en datos, mejorar gradualmente el rendimiento de los modelos predictivos y tomar decisiones basadas en esos datos. El aprendizaje automático no solo es cada vez más importante en la investigación de ciencia computacional, sino que juega un papel cada vez más importante en nuestra vida diaria. Gracias al aprendizaje automático, disfrutamos de filtros potentes para el correo no deseado, *software* práctico de reconocimiento de voz y texto, motores de búsqueda fiables, desafiantes programas para jugar al ajedrez y, esperemos que muy pronto, eficientes coches de conducción autónoma.

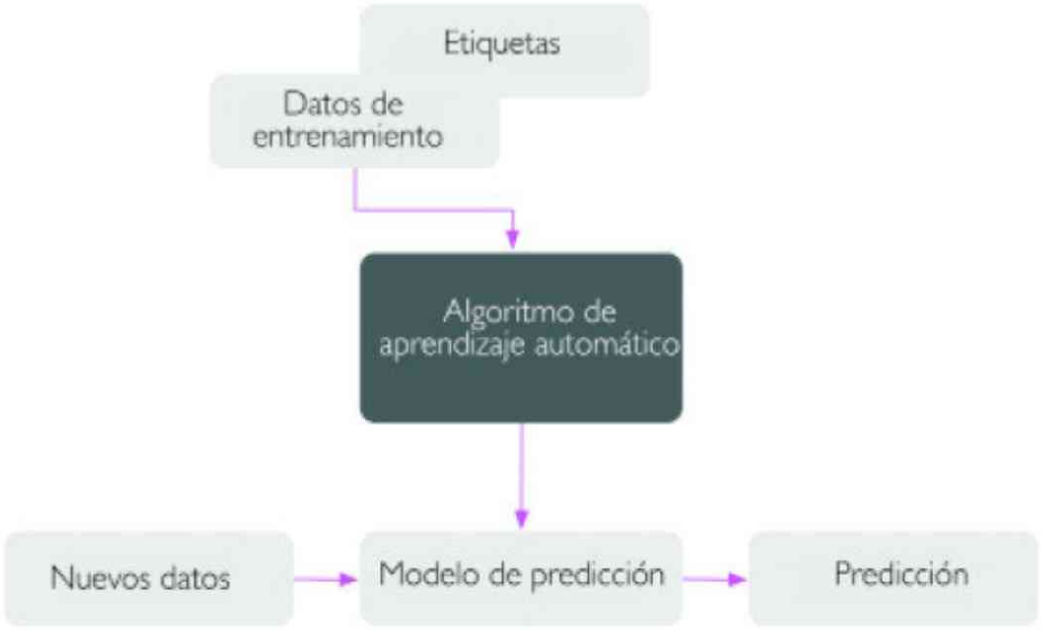
Los tres tipos de aprendizaje automático

En esta sección, echaremos un vistazo a los tres tipos de aprendizaje automático: **aprendizaje supervisado**, **aprendizaje no supervisado** y **aprendizaje reforzado**. Vamos a aprender las diferencias fundamentales entre los tres tipos distintos de aprendizaje y, mediante ejemplos conceptuales, desarrollaremos una intuición para los ámbitos de problemas prácticos donde pueden ser aplicados:



Hacer predicciones sobre el futuro con el aprendizaje supervisado

El objetivo principal del aprendizaje supervisado es aprender un modelo, a partir de datos de entrenamiento etiquetados, que nos permite hacer predicciones sobre datos futuros o no vistos. Aquí, el término **supervisado** se refiere a un conjunto de muestras donde las señales de salida deseadas (etiquetas) ya se conocen.



Considerando el ejemplo del filtro de correo no deseado, podemos entrenar un modelo utilizando un algoritmo de aprendizaje automático supervisado en un cuerpo de correos electrónicos etiquetados –correos que están correctamente marcados como «correo no deseado» o como «no correo no deseado»– para predecir si un nuevo correo electrónico pertenece a una u otra categoría. Una tarea de aprendizaje supervisado con etiquetas de clase discretas, como en el ejemplo anterior del filtro de correo no deseado, también se conoce como **tarea de clasificación**. Otra subcategoría del aprendizaje supervisado es la **regresión**, donde la señal resultante es un valor continuo.

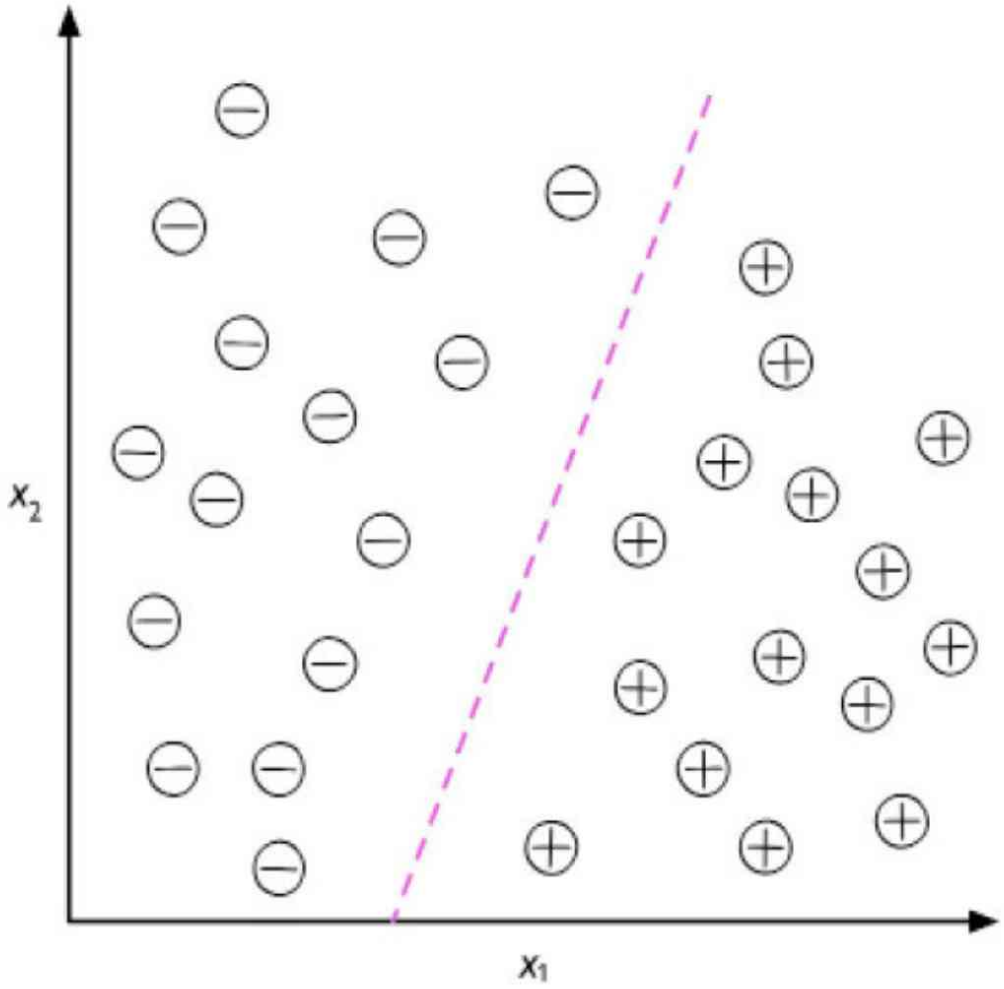
Clasificación para predecir etiquetas de clase

La clasificación es una subcategoría del aprendizaje supervisado cuyo objetivo es predecir las etiquetas de clase categórica de nuevas instancias, basadas en observaciones pasadas. Estas etiquetas de clase son discretas, valores desordenados que se pueden entender como membresías grupales de las instancias. El ejemplo que hemos mencionado anteriormente de la detección de correo no deseado representa un típico ejemplo de una tarea de clasificación binaria, donde el algoritmo de aprendizaje automático aprende un conjunto de reglas para distinguir entre dos posibles clases: mensajes que son o no son correo no deseado.

Sin embargo, el conjunto de etiquetas de clase no tiene que ser de naturaleza binaria. El modelo predictivo aprendido mediante un algoritmo de aprendizaje supervisado puede asignar cualquier etiqueta de clase que se presente en el conjunto de datos de entrenamiento a una nueva instancia sin etiqueta.

Un ejemplo típico de una tarea de **clasificación multiclase** es el reconocimiento de un carácter manuscrito. Aquí, podemos recoger un conjunto de datos de entrenamiento que consiste en múltiples ejemplos manuscritos de cada letra del alfabeto. Ahora, si un usuario proporciona un nuevo carácter manuscrito desde un dispositivo de entrada, nuestro modelo predictivo será capaz de predecir la letra correcta del alfabeto con cierta precisión. Sin embargo, nuestro sistema de aprendizaje automático no sería capaz de reconocer de forma correcta ningún dígito del cero al nueve, por ejemplo, si no formaran parte de nuestro conjunto de datos de entrenamiento.

La siguiente figura ilustra el concepto de una tarea de clasificación binaria que da 30 muestras de entrenamiento; 15 de estas muestras están etiquetadas como clase negativa (signo menos) y otras 15 como clase positiva (signo más). En este caso, nuestro conjunto de datos es bidimensional, lo que significa que cada muestra tiene dos valores asociados: x_1 y x_2 . Ahora, podemos utilizar un algoritmo de aprendizaje automático supervisado para aprender una regla –el límite de decisión está representado con una línea discontinua– que puede separar las dos clases y clasificar nuevos datos dentro de cada categoría dados sus valores de x_1 y x_2 :



https://yolibrospdf.com/programacion.html

Regresión para predecir resultados continuos

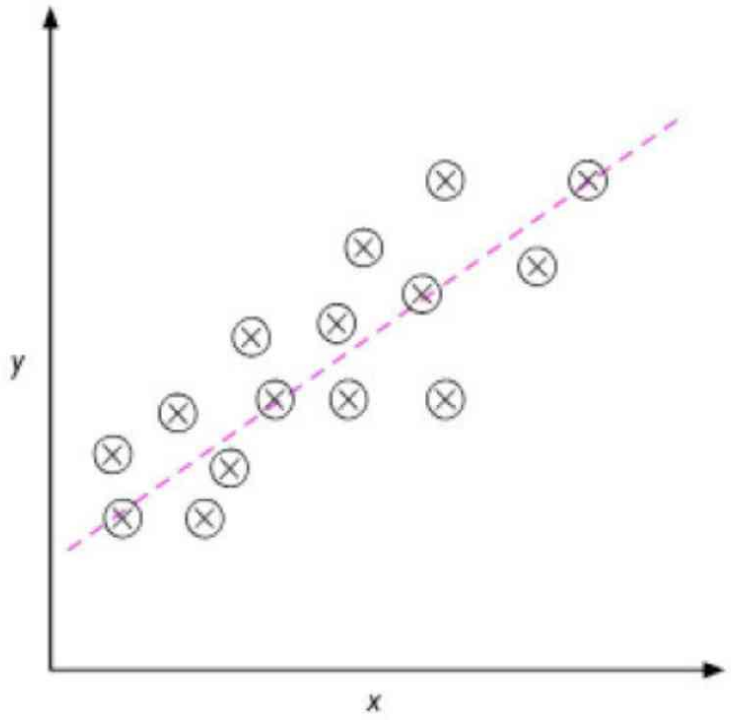
En la sección anterior hemos aprendido que la tarea de clasificación consiste en asignar etiquetas categóricas y sin orden a instancias. Un segundo tipo de aprendizaje supervisado es la predicción de resultados continuos, también conocida como **análisis de regresión**. En el análisis de regresión, tenemos un número de variables predictoras **(explicativas)** y una variable de respuesta continua (**resultado** o **destino**), y tenemos que encontrar una relación entre estas variables que nos permita predecir un resultado.

Por ejemplo, supongamos que queremos predecir los resultados del examen de selectividad de matemáticas de nuestros alumnos. Si existe una relación entre el tiempo que han pasado estudiando para la prueba y los resultados finales, podríamos utilizarla como dato de entrenamiento para aprender un modelo que utilice el tiempo de estudio para predecir los resultados de la prueba de futuros estudiantes que deseen pasar este examen.

El término *regresión* fue ideado por Francis Galton en su artículo *Regression towards Mediocrity in Hereditary Stature* [Regresión hacia la mediocridad en estatura hereditaria] en 1886. Galton describió el fenómeno biológico según el cual la variación de altura en una población no aumenta con el tiempo. Él observó que la altura de los padres no pasa a los hijos, pero que, en cambio, la altura de los hijos está retrocediendo hacia la media de la población.

La siguiente figura ilustra el concepto de regresión lineal. Dada una variable predictora x y una variable de respuesta y, aplicamos una línea fina a este

dato, que minimiza la distancia –normalmente, la distancia cuadrada de promedio– entre los puntos de muestra y la línea aplicada. Ahora podemos utilizar la intersección y la pendiente aprendidas de este dato para predecir la variable de resultado del nuevo dato:



Resolver problemas interactivos con aprendizaje reforzado

Otro tipo de aprendizaje automático es el **aprendizaje reforzado**. En este tipo de aprendizaje, el objetivo es desarrollar un sistema **(agente)** que mejore su rendimiento basado en interacciones con el entorno. Como la información sobre el estado actual del entorno normalmente también incluye una **señal de recompensa**, podemos pensar en el aprendizaje reforzado como un campo relacionado con el aprendizaje supervisado. Sin embargo, en el aprendizaje reforzado este *feedback* no es el valor o la etiqueta correctos sobre el terreno, sino una medida de cómo ha sido medida la acción por parte de una función de recompensa. A través de su interacción con el entorno, un agente puede utilizar el aprendizaje reforzado para aprender una serie de acciones que maximicen esta recompensa mediante un enfoque experimental de ensayo-error o una planificación deliberativa.

Un conocido ejemplo de aprendizaje reforzado es un motor de ajedrez. Aquí, el agente elige entre una serie de movimientos según el estado del tablero (el entorno), y la recompensa se puede definir como **«ganas»** o **«pierdes»** al final del juego:

Existen diferentes subtipos de aprendizaje reforzado. Sin embargo, un esquema general es que el agente en aprendizaje reforzado intenta maximizar la recompensa mediante una serie de interacciones con el entorno. Cada estado puede estar asociado a una recompensa positiva o negativa, y una recompensa se puede definir como el logro de un objetivo general (como

ganar o perder una partida de ajedrez). Por ejemplo, en ajedrez, el resultado de cada movimiento podría ser un estado distinto del entorno. Para explorar un poco más el ejemplo del ajedrez, pensemos en ciertas jugadas del tablero asociadas a un evento positivo (por ejemplo, eliminar una pieza del contrincante o amenazar a la reina). Sin embargo, otras jugadas están asociadas a un evento negativo (como perder una pieza para el contrincante en el siguiente turno). Ahora, no todos los turnos dan como resultado la eliminación de una pieza del tablero, y el aprendizaje reforzado se centra en aprender las series de pasos maximizando una recompensa basada en el *feedback* inmediato y diferido.

Aunque esta sección ofrece una visión básica del aprendizaje reforzado, ten en cuenta que las aplicaciones de este tipo de aprendizaje están fuera del alcance de este libro, que prioriza la clasificación, el análisis de regresión y el agrupamiento.

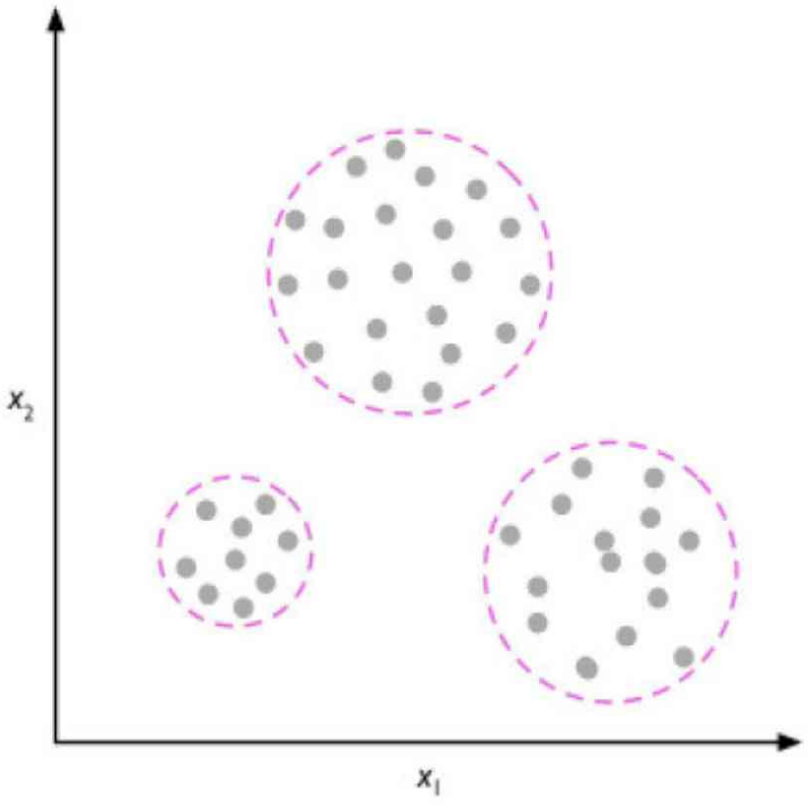
Descubrir estructuras ocultas con el aprendizaje sin supervisión

En el aprendizaje supervisado, cuando entrenamos nuestro modelo sabemos la respuesta correcta de antemano, y en el reforzado definimos una medida de recompensa para acciones concretas mediante el agente. Sin embargo, en el aprendizaje sin supervisión tratamos con datos sin etiquetar o datos de estructura desconocida. Con las técnicas de aprendizaje sin supervisión, podemos explorar la estructura de nuestros datos para extraer información significativa sin la ayuda de una variable de resultado conocida o una función de recompensa.

Encontrar subgrupos con el agrupamiento

El **agrupamiento** es una técnica exploratoria de análisis de datos que nos permite organizar un montón de información en subgrupos significativos ***(clústers)*** sin tener ningún conocimiento previo de los miembros del grupo. Cada *clúster* que surge durante el análisis define un grupo de objetos que comparten un cierto grado de semejanza pero difieren de los objetos de otros *clústers*, razón por la cual el agrupamiento también se denomina a veces **clasificación sin supervisión**. El agrupamiento es una excelente técnica para estructurar información y derivar relaciones significativas de los datos. Por ejemplo, permite a los vendedores descubrir grupos de clientes basados en sus intereses, con el fin de desarrollar programas de *marketing* exclusivos.

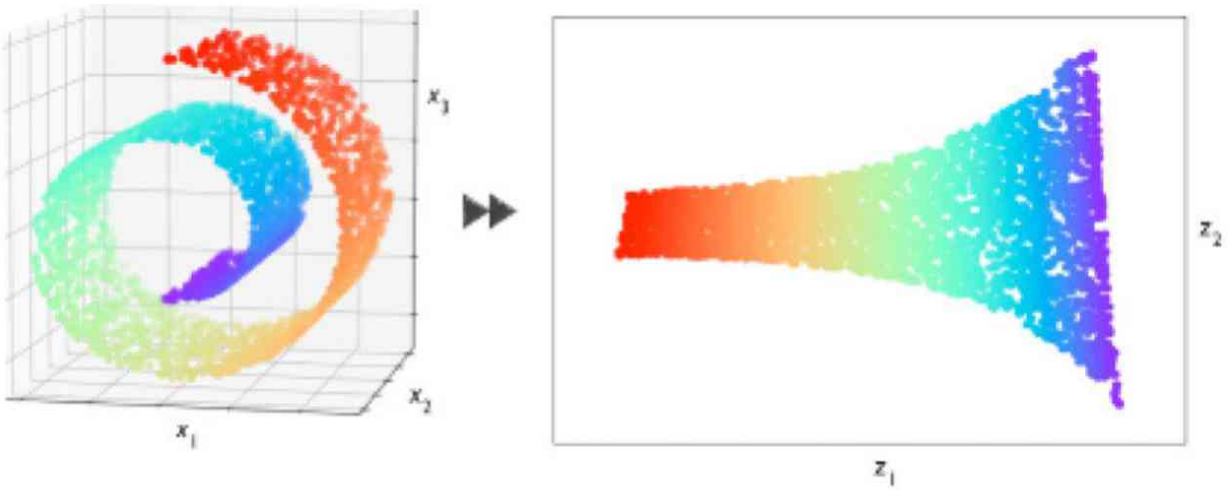
La siguiente figura muestra cómo se puede aplicar el agrupamiento para organizar datos sin etiquetar en tres grupos distintos, basados en la similitud de sus características x_1 y x_2 :



Reducción de dimensionalidad para comprimir datos

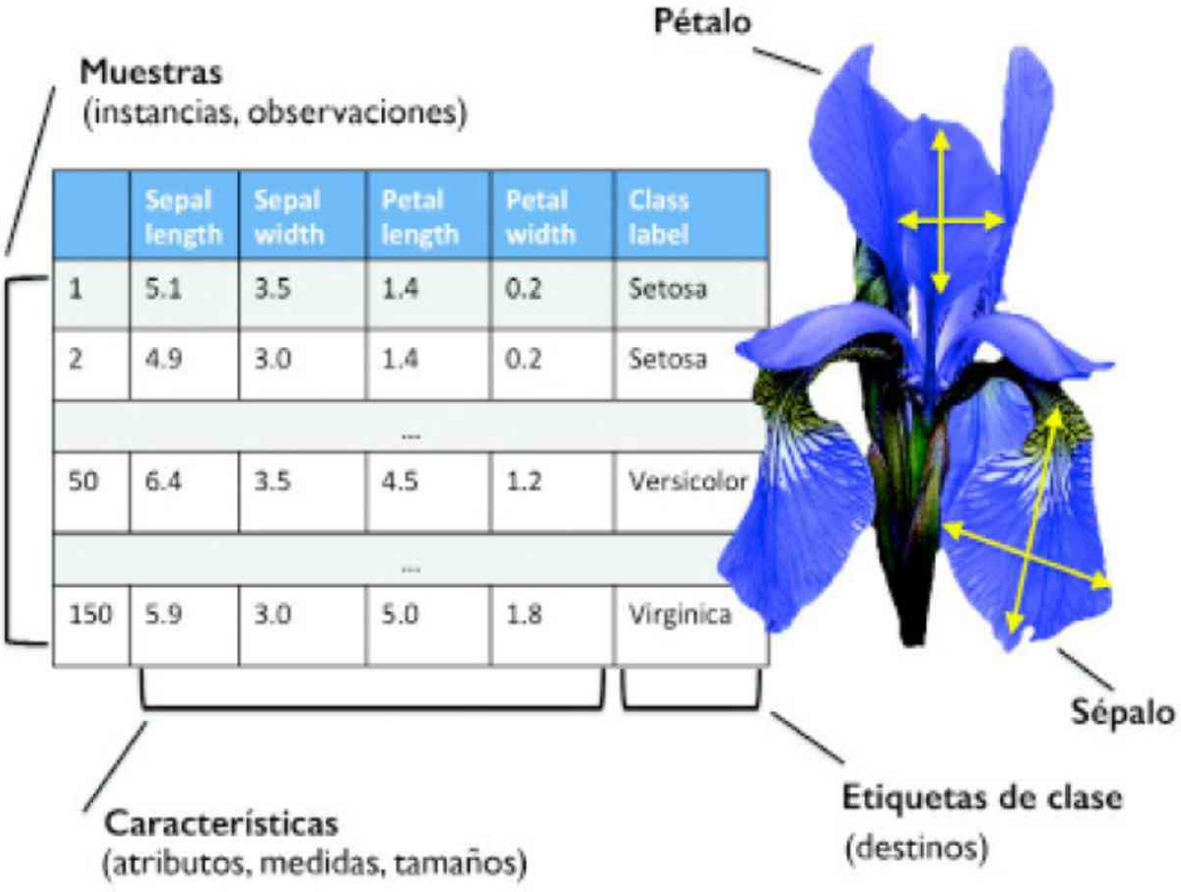
Otro subcampo del aprendizaje sin supervisión es la **reducción de dimensionalidad**. Muchas veces trabajamos con datos de alta dimensionalidad (cada observación muestra un elevado número de medidas), lo cual puede suponer un reto para el espacio de almacenamiento limitado y el rendimiento computacional de los algoritmos del aprendizaje automático. La reducción de dimensionalidad sin supervisión es un enfoque utilizado con frecuencia en el preprocesamiento de características para eliminar ruido de los datos; también puede degradar el rendimiento predictivo de ciertos algoritmos y comprimir los datos en un subespacio dimensional más pequeño, manteniendo la mayor parte de la información importante.

A veces, la reducción de dimensionalidad también puede ser útil para visualizar datos; por ejemplo, un conjunto de características dimensionales pueden ser proyectadas en un espacio de características de una, dos o tres dimensiones para visualizarlas mediante gráficos de dispersión o histogramas 2D o 3D. Las siguientes figuras muestran un ejemplo donde la reducción de dimensionalidad no lineal se ha aplicado para comprimir un brazo de gitano tridimensional en un subespacio con características 2D:



Introducción a la terminología básica y las notaciones

Ahora que ya hemos tratado las tres categorías principales de aprendizaje automático –supervisado, sin supervisión y reforzado–, vamos a echar un vistazo a la terminología básica que utilizaremos en este libro. La tabla siguiente muestra un extracto del conjunto de datos Iris, un ejemplo clásico en el campo del aprendizaje automático. El conjunto de datos Iris contiene las medidas de 150 flores iris de tres especies distintas: Setosa, Versicolor y Virginica. Cada muestra de flor representa una fila de nuestro conjunto de datos y las medidas de la flor en centímetros se almacenan en columnas, que también denominamos **características** del conjunto de datos:



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Para que la notación sea simple a la vez que eficiente, utilizaremos algunos

de los términos básicos de álgebra lineal. En los siguientes capítulos, utilizaremos una matriz y una notación vectorial para referirnos a nuestros datos. Seguiremos la convención común para representar cada muestra como una fila independiente en una matriz de características **X**, donde cada característica se almacena en una columna independiente.

Así, el conjunto de datos Iris que contiene 150 muestras y cuatro características también se puede escribir como una matriz 150×4 $\boldsymbol{X} \in \mathbb{R}^{150 \times 4}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

Para el resto del libro, si no se indica de otro modo, utilizaremos el superíndice *i* para indicar la muestra de entrenamiento *i*, y el subíndice *j* para indicar la dimensión *j* del conjunto de datos de entrenamiento. Utilizamos letras en negrita y minúsculas para referirnos a vectores ($\boldsymbol{x} \in \mathbb{R}^{n \times 1}$) y letras en negrita y mayúsculas para hablar de matrices ($\boldsymbol{X} \in \mathbb{R}^{n \times m}$). Para referirnos a elementos individuales en un vector o matriz, escribimos las letras en cursiva ($x^{(n)}$ o $x_{(m)}^{(n)}$, respectivamente).

Por ejemplo, x_1^{150} se refiere a la primera dimensión de las 150 muestras de flores, *largo de sépalo*. Así, cada fila de la matriz de características representa una instancia de flor y puede ser escrita como un vector de fila de cuatro dimensiones $\boldsymbol{x}^{(i)} \in \mathbb{R}^{1 \times 4}$:

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

Y cada dimensión de características es un vector de columna de 150 dimensiones $\boldsymbol{x}_j \in \mathbb{R}^{150 \times 1}$. Por ejemplo:

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

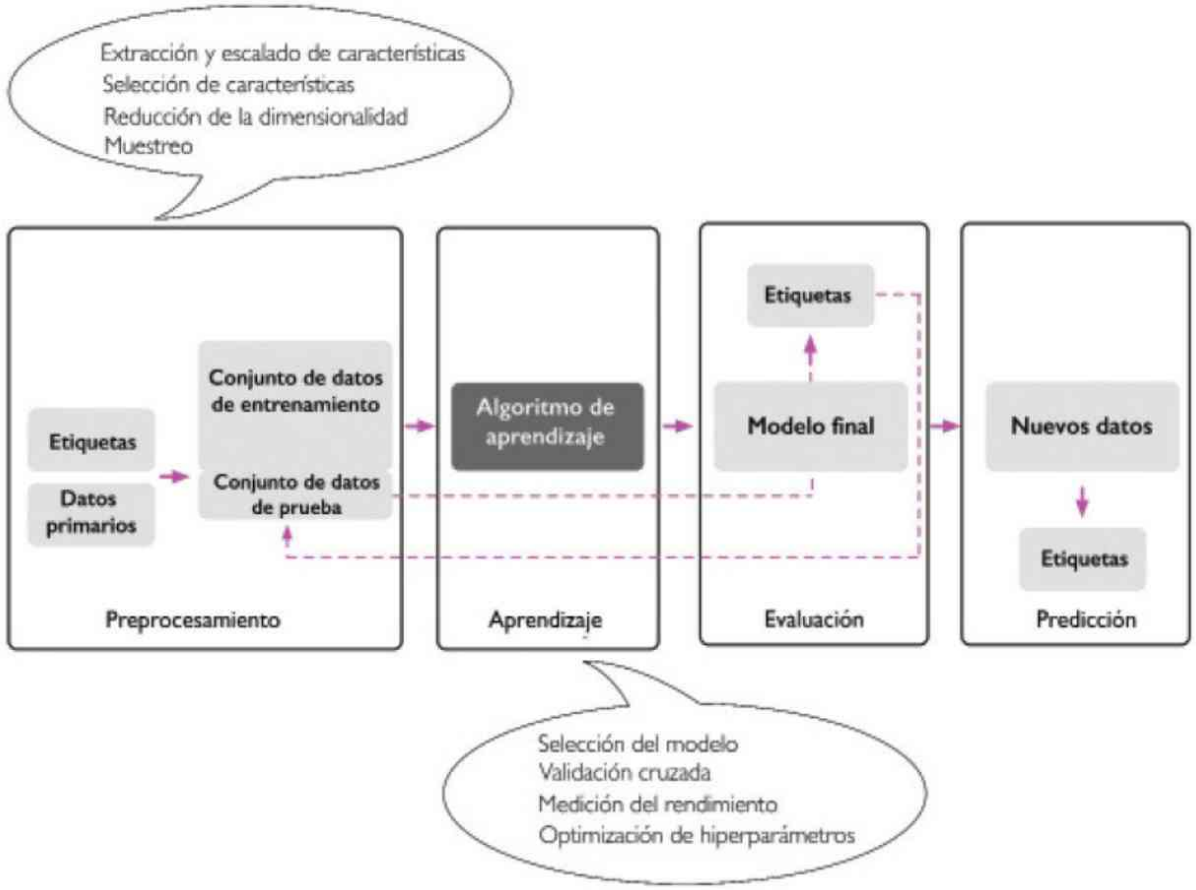
De forma similar, almacenamos las

variables de destino (aquí, etiquetas de clase) como un vector de columna de 150 dimensiones:

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left(y \in \{\text{Setosa, Versicolor, Virginica}\} \right)$$

Una hoja de ruta para crear sistemas de aprendizaje automático

En secciones anteriores, hemos hablado de los conceptos básicos del aprendizaje automático y de los tres tipos distintos de aprendizaje. En esta sección, hablaremos de las otras partes importantes del sistema de aprendizaje automático que acompañan al algoritmo de aprendizaje. El siguiente diagrama muestra un flujo de trabajo típico para el uso del aprendizaje automático en modelado predictivo, que trataremos en las siguientes subsecciones:



Preprocesamiento: Dar forma a los datos

Vamos a empezar hablando de la hoja de ruta para crear sistemas de aprendizaje automático. No es habitual que los datos primarios se presenten en la forma necesaria para un rendimiento óptimo del algoritmo de aprendizaje. Así, el preprocesamiento de los datos es uno de los pasos más importantes en cualquier aplicación de aprendizaje automático. Si tomamos como ejemplo el conjunto de datos de flores Iris de la sección anterior, podemos pensar en los datos primarios como una serie de imágenes de flores de las cuales queremos extraer características significativas. Estas características útiles pueden ser el color, el tono, la intensidad de las flores, o la altura, la longitud y anchura de la flor. Hay algoritmos de aprendizaje automático que, además, necesitan que las características seleccionadas tengan el mismo tamaño para conseguir un rendimiento óptimo, el cual normalmente se consigue transformando las características en el rango [0, 1] o con una distribución normal estándar con media cero y variación unitaria, como veremos más adelante.

Algunas de las características seleccionadas pueden estar altamente relacionadas y, por tanto, pueden ser redundantes hasta un cierto punto. En estos casos, las técnicas de reducción de dimensionalidad son muy útiles para comprimir las características en un subespacio dimensional más pequeño. Reducir la dimensionalidad de nuestro espacio de características tiene la ventaja de que se requiere menos espacio de almacenamiento y el algoritmo de aprendizaje funciona mucho más rápido. En algunos casos, la reducción de dimensionalidad también puede mejorar el rendimiento predictivo de un modelo si el conjunto de datos contiene un gran número de características irrelevantes (o ruido), es decir, si el conjunto de datos tiene una relación baja entre señal y ruido.

Para determinar si nuestro algoritmo de aprendizaje automático no solo funciona bien en el conjunto de entrenamiento sino que también se generaliza

en datos nuevos, también nos interesa dividir de forma aleatoria el conjunto de datos en un conjunto de prueba y de entrenamiento individual. Utilizamos el conjunto de entrenamiento para entrenar y optimizar nuestro modelo de aprendizaje automático, al tiempo que mantenemos el conjunto de prueba hasta el final para evaluar el modelo final.

Entrenar y seleccionar un modelo predictivo

Como veremos en capítulos próximos, se han desarrollado diferentes tipos de algoritmos de aprendizaje automático para resolver distintas tareas problemáticas. Un punto importante que se puede resumir de los famosos teoremas de *No hay almuerzo gratis* de David Wolpert es que no podemos aprender «gratis». Algunas publicaciones más importantes son *The Lack of A Priori Distinctions Between Learning Algorithms* [La falta de distinciones *a priori* entre los algoritmos de aprendizaje], D. H. Wolpert (1996); *No free lunch theorems for optimization* [Teoremas de no hay almuerzo gratis para la optimización], D. H. Wolpert y W.G. Macready (1997). Intuitivamente, podemos relacionar este concepto con la popular frase: «Si tu única herramienta es un martillo, tiendes a tratar cada problema como si fuera un clavo» (Abraham Maslow, 1966). Por ejemplo, cada algoritmo de clasificación tiene sus sesgos inherentes, y ninguna clasificación individual es superior si no hacemos suposiciones sobre la tarea. En la práctica, resulta esencial comparar como mínimo un puñado de algoritmos distintos para entrenar y seleccionar el mejor modelo de rendimiento. Pero antes de comparar los diferentes modelos, debemos decidir una unidad para medir el rendimiento. Una unidad de medida que se utiliza con frecuencia es la precisión de la clasificación, que se define como la proporción de instancias clasificadas correctamente.

Una cuestión legítima que podemos preguntarnos es: «¿Cómo podemos saber qué modelo funciona bien en el conjunto de datos final y los datos reales si no utilizamos este conjunto de prueba para la selección del modelo, pero sí lo mantenemos para la evolución del modelo final?». Para abordar el problema incluido en esta cuestión, se pueden utilizar diferentes técnicas de validación cruzada, donde el conjunto de datos de entrenamiento se divide en subconjuntos de validación y entrenamiento para estimar el rendimiento de generalización del modelo. Al final, no podemos esperar que los parámetros

predeterminados de los diferentes algoritmos de aprendizaje proporcionados por las librerías de los programas sean óptimos para nuestra tarea problemática concreta. Por lo tanto, utilizaremos a menudo técnicas de optimización de hiperparámetros que nos ayudarán, en próximos capítulos, a afinar el rendimiento de nuestro modelo. Intuitivamente, podemos pensar en dichos hiperparámetros como parámetros que no se aprenden de los datos sino que representan los botones de un modelo que podemos girar para mejorar su rendimiento. Todo esto quedará más claro en capítulos posteriores, donde veremos ejemplos reales.

Evaluar modelos y predecir instancias de datos no vistos

Después de haber seleccionado un modelo instalado en el conjunto de datos de entrenamiento, podemos utilizar el conjunto de datos de prueba para estimar cómo funciona con los datos no vistos para estimar el error de generalización. Si su rendimiento nos satisface, ya podemos utilizar este modelo para predecir nuevos y futuros datos. Es importante observar que los parámetros para los procedimientos mencionados anteriormente, como el escalado de características y la reducción de dimensionalidad, solo pueden obtenerse a partir de conjuntos de datos de entrenamiento, y que los mismos parámetros vuelven a aplicarse más tarde para transformar el conjunto de datos de prueba, así como cualquier nueva muestra de datos. De otro modo, el rendimiento medido en los datos de prueba puede ser excesivamente optimista.

Utilizar Python para el aprendizaje automático

Python es uno de los lenguajes de programación más populares para la ciencia de datos y, por ello, cuenta con un elevado número de útiles librerías complementarias desarrolladas por sus excelentes desarrolladores y su comunidad de código abierto.

Aunque el rendimiento de los lenguajes interpretados –como Python– para tareas de cálculo intensivo es inferior al de los lenguajes de bajo nivel, se han desarrollado librerías como NumPy y SciPy sobre implementaciones de C y Fortran de capa inferior para operaciones rápidas y vectorizadas en matrices multidimensionales.

Para tareas de programación de aprendizaje automático, haremos referencia sobre todo a la librería scikit-learn, que actualmente es una de las librerías de aprendizaje automático de código abierto más popular y accesible.

Instalar Python y sus paquetes desde el Python Package Index

Python está disponible para los tres sistemas operativos principales –Microsoft Windows, macOS y Linux– y tanto el instalador como la documentación se pueden descargar desde el sitio web oficial de Python: https://www.python.org.

Este libro está escrito para Python versión 3.5.2 o posterior, y es recomendable que utilices la versión más reciente de Python 3 que esté disponible actualmente, aunque la mayoría de los ejemplos de código también son compatibles con Python 2.7.13 o superior. Si decides utilizar Python 2.7 para ejecutar los ejemplos de código, asegúrate de que conoces las diferencias principales entre ambas versiones. Puedes consultar un buen resumen de las diferencias entre Python 3.5 y 2.7 en https://wiki.python.org/moin/Python2orPython3.

Los paquetes adicionales que se utilizarán en este libro se pueden instalar mediante el programa de instalación **pip**, que forma parte de la librería

estándar de Python desde la versión 3.3. Puedes encontrar más información sobre el **pip** en https://docs.python.org/3/installing/index.html.

Una vez hemos instalado Python con éxito, podemos ejecutar el **pip** desde el terminal para instalar los paquetes de Python adicionales:

```
pip install SomePackage
```

Los paquetes ya instalados pueden ser actualizados con el comando **--upgrade**:

```
pip install SomePackage --upgrade
```

Utilizar la distribución y el gestor de paquetes Anaconda de Python

Una distribución de Python alternativa muy recomendada para cálculo científico es Anaconda, de Continuum Analytics. Anaconda es una distribución gratuita (incluso para uso comercial) de Python preparada para la empresa, que incluye todos los paquetes de Python esenciales para la ciencia de datos, matemáticas e ingeniería en una distribución multiplataforma fácil de usar.

El instalador de Anaconda se puede descargar desde http://continuum.io/down-loads y hay disponible una guía de inicio rápido de Anaconda en https://conda.io/docs/test-drive.html.

Tras haber instalado Anaconda con éxito, podemos instalar los nuevos paquetes de Python mediante el siguiente comando:

conda install SomePackage

Los paquetes existentes se pueden actualizar mediante el siguiente comando:

conda update SomePackage

Paquetes para cálculo científico, ciencia de datos y aprendizaje automático

En este libro, utilizaremos principalmente matrices multidimensionales de NumPy para almacenar y manipular datos. De forma ocasional, utilizaremos pandas, una librería creada sobre NumPy que proporciona herramientas de manipulación de datos de alto nivel que permiten trabajar con datos tabulados de un modo más conveniente. Para aumentar nuestra experiencia de aprendizaje y visualizar datos cuantitativos, que a menudo es extremadamente útil para dar sentido a todo esto de manera intuitiva, utilizaremos la librería Matplotlib, que se puede personalizar en muchos aspectos.

Las versiones de los principales paquetes de Python que se han utilizado para escribir este libro son las que aparecen en la siguiente lista. Asegúrate de que la versión de los paquetes que has instalado sea igual o superior a estas versiones para garantizar que los ejemplos de código funcionen correctamente:

- NumPy 1.12.1
- SciPy 0.19.0
- scikit-learn 0.18.1
- Matplotlib 2.0.2
- pandas 0.20.1

Resumen

En este capítulo, hemos explorado el aprendizaje automático desde un nivel muy alto y nos hemos familiarizado con el panorama general y los principales conceptos que vamos a explorar con mayor detalle en los próximos capítulos. Hemos aprendido que el aprendizaje supervisado está compuesto por dos importantes subcampos: clasificación y regresión. Mientras que los modelos de clasificación nos permiten categorizar objetos en clases conocidas, podemos utilizar el análisis de regresión para predecir el resultado continuo de variables de destino. El aprendizaje sin supervisión no solo ofrece técnicas útiles para descubrir estructuras en datos sin etiquetar, sino que también puede ser útil para la compresión de datos en las etapas de preprocesamiento de características. Nos hemos referido brevemente a la hoja de ruta típica para aplicar el aprendizaje automático a problemas concretos, que usaremos como base para discusiones más profundas y ejemplos prácticos en los siguientes capítulos. Además, hemos preparado nuestro entorno Python e instalado y actualizado los paquetes necesarios para estar listos para ver en acción el aprendizaje automático.

Más adelante en este libro, además del aprendizaje automático en sí mismo, también presentaremos diferentes técnicas para preprocesar nuestros conjuntos de datos, que nos ayudarán a conseguir el mejor rendimiento de los distintos algoritmos de aprendizaje automático. Si bien cubriremos los algoritmos de clasificación de forma bastante extensa en todo el libro, también exploraremos diferentes técnicas para el análisis de regresión y la agrupación. Tenemos un emocionante viaje por delante, en el que descubriremos potentes técnicas en el amplio campo del aprendizaje automático. Sin embargo, nos acercaremos al aprendizaje automático paso a paso, generando nuestro conocimiento de forma gradual a lo largo de los capítulos que componen este libro. En el capítulo siguiente, empezaremos este viaje implementando uno

de los algoritmos de aprendizaje automático para clasificación, que nos prepara para el *Capítulo 3, Un recorrido por los clasificadores de aprendizaje automático con scikit-learn*, donde nos acercaremos a algoritmos de aprendizaje automático más avanzados con la librería de código abierto scikit-learn.

2

Entrenar algoritmos simples de aprendizaje automático para clasificación

En este capítulo, utilizaremos dos de los primeros algoritmos de aprendizaje automático descritos algorítmicamente para clasificación: el perceptrón y las neuronas lineales adaptativas. Empezaremos implementando un perceptrón paso a paso en Python y entrenándolo para que clasifique diferentes especies de flores en el conjunto de datos Iris. Esto nos ayudará a entender el concepto de algoritmos de aprendizaje automático para clasificación y cómo pueden ser implementados de forma eficiente en Python.

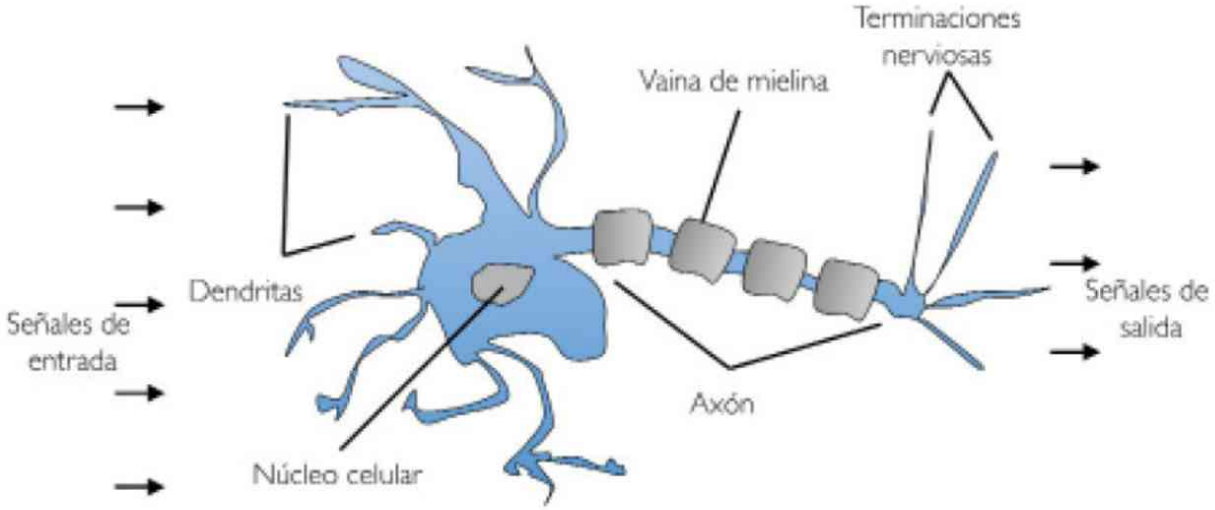
Tratar los conceptos básicos de la optimización utilizando neuronas lineales adaptativas sentará las bases para el uso de clasificadores más potentes con la librería de aprendizaje automático scikit-learn, como veremos en el *Capítulo 3, Un recorrido por los clasificadores de aprendizaje automático con scikit-learn*.

Los temas que trataremos en este capítulo son los siguientes:

- Crear una intuición para algoritmos de aprendizaje automático.
- Utilizar pandas, NumPy y Matplotlib para leer, procesar y visualizar datos.
- Implementar algoritmos de clasificación lineal en Python.

Neuronas artificiales: un vistazo a los inicios del aprendizaje automático

Antes de hablar con más detalle del perceptrón y de los algoritmos relacionados, echemos un vistazo a los comienzos del aprendizaje automático. Para tratar de entender cómo funciona el cerebro biológico, para diseñar la Inteligencia Artificial, Warren McCullock y Walter Pitts publicaron, en 1943, el primer concepto de una célula cerebral simplificada, la denominada neurona **McCullock-Pitts (MCP)**, recogido en su libro *A Logical Calculus of the Ideas Immanent in Nervous Activity* [Un cálculo lógico de las ideas inmanentes en la actividad nerviosa], *W. S. McCulloch* y *W. Pitts, Bulletin of Mathematical Biophysics* [Boletín de Biofísica Matemática], 5(4): 115-133, *1943*. Las neuronas son células nerviosas interconectadas en el cerebro que participan en el proceso y la transmisión de señales eléctricas y químicas, como se ilustra en la siguiente figura:



McCullock y Pitts describieron una célula nerviosa como una simple puerta lógicacon salidas binarias; múltiples señales llegan a las dendritas, a continuación se integran en el cuerpo de la célula y, si la señal acumulada supera un umbral determinado, se genera una señal de salida que será transmitida por el axón.

Solo unos años después, Frank Rosenblatt publicó el primer concepto de la regla de aprendizaje del perceptrón, basado en el modelo de la neurona MCP, en *The Perceptron: A Perceiving and Recognizing Automaton* [El perceptrón: un autómata de percepción y reconocimiento], *F. Rosenblatt, Cornell Aeronautical Laboratory, 1957*). Con esta regla del perceptrón, Rosenblatt propuso un algoritmo que podía automáticamente aprender los coeficientes de peso óptimo que luego se multiplican con las características de entrada para tomar la decisión de si una neurona se activa o no. En el contexto del aprendizaje supervisado y la clasificación, un algoritmo como este podría utilizarse para predecir si una muestra pertenece a una clase o a otra.

La definición formal de una neurona artificial

De un modo más formal, para simplificar, podemos situar la idea de las **neuronas artificiales** en el contexto de una tarea de clasificación binaria donde hacemos referencia a nuestras dos clases como 1 (clase positiva) y -1 (clase negativa). También podemos definir una función de decisión ($\phi(z)$) que toma una combinación lineal de determinados valores de entrada **x** y un vector de peso correspondiente **w**, donde z es la denominada entrada de red $z = w_1 x_1 + \ldots + w_m x_m$:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Ahora, si la entrada de red de una muestra concreta $\boldsymbol{x}^{(i)}$ es mayor que un umbral definido θ , predecimos de otro modo la clase *1* y la clase *-1*. En el algoritmo de perceptrón, la función de decisión $\phi(\cdot)$ es una variante de una **función escalón unitario**:

$$\phi(z) = \begin{cases} 1 \ if \ z \geq 0 \\ -1 \ otherwise \end{cases}$$

Para simplificar, podemos traer el umbral θ al lado izquierdo de la ecuación y definir un peso cero como $w_0 = -\theta$ y $x_0 = 1$, por lo que escribimos z de un modo más compacto:

$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

Y:

$$\phi(z) = \begin{cases} 1 \ if \ z \geq 0 \\ -1 \ otherwise \end{cases}$$

En la literatura del aprendizaje automático, el umbral negativo, o peso $w_0 = -\theta$, se denomina habitualmente «sesgo» (***bias***, en inglés).

En las siguientes secciones, usaremos con frecuencia notaciones básicas de álgebra lineal. Por ejemplo, abreviaremos la suma de los productos de los valores en **x** y **w** con un producto escalar vectorial, donde el superíndice ^{T} se refiere a **trasposición**, que es una operación que transforma una vector columna en un vector fila y viceversa.

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} \boldsymbol{x}_j \boldsymbol{w}_j = \boldsymbol{w}^T \boldsymbol{x}$$

Por ejemplo:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$$

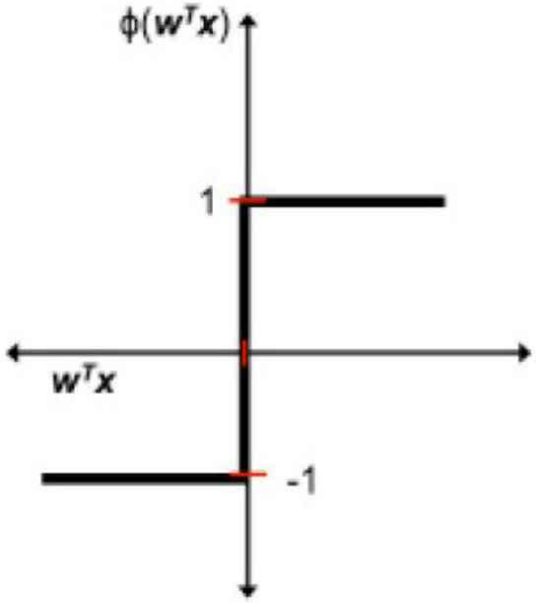
Además, la operación de trasposición también se puede aplicar a matrices para reflejarlas sobre su diagonal, por ejemplo:

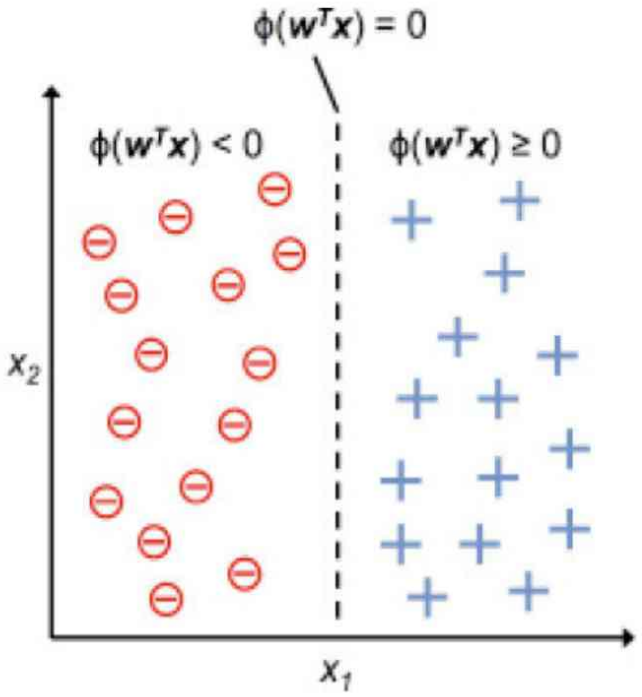
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

En este libro, utilizaremos solo conceptos muy básicos de álgebra lineal; sin embargo, si necesitas un repaso rápido, puedes echar un vistazo a la excelente obra de Zico Kolter *Linear Algebra Review and Reference* [Revisión y referencia de álgebra lineal], disponible de forma gratuita en http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf.

La siguiente figura ilustra cómo la función de decisión del perceptrón (sub-figura izquierda) comprime la entrada de red $z = \boldsymbol{w}^T \boldsymbol{x}$ en una salida binaria

(**-1** o **1**) y cómo se puede utilizar para discriminar entre dos clases separables linealmente (subfigura derecha):


$\phi(w^T x)$
1
$w^T x$
-1


$\phi(w^T x) = 0$
$\phi(w^T x) < 0$
$\phi(w^T x) \geq 0$
x_2
x_1

La regla de aprendizaje del perceptrón

La idea general que hay detrás de la neurona MCP y del modelo de perceptrón *umbralizado* de Rosenblatt es utilizar un enfoque reduccionista para imitar cómo trabaja una simple neurona en el cerebro: si se *excita* o si no. Así, la regla del perceptrón inicial de Rosenblatt es bastante sencilla y se puede resumir en los siguientes pasos:

1. Iniciar los pesos a 0 o a números aleatorios más pequeños.
2. Para cada muestra de entrenamiento $\boldsymbol{x}^{(i)}$:
    a. Calcular el valor de salida $\hat{y}$.
    b. Actualizar los pesos.

Aquí, el valor de salida es la etiqueta de clase predicha por la función escalón unitario que hemos definido anteriormente, y la actualización simultánea de cada peso w_j en el vector peso **w** se puede escribir formalmente como:

$$w_j := w_j + \Delta w_j$$

El valor de Δw_j , que se utiliza para actualizar el peso w_j , se calcula mediante la regla de aprendizaje del perceptrón:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

donde η es el **rango de aprendizaje** (normalmente una constante entre 0.0 y 1.0), $y^{(i)}$ es la **etiqueta de clase verdadera** de la muestra de entrenamiento *i*, y $\hat{y}^{(i)}$ es la **etiqueta de clase predicha**. Es importante observar que todos los pesos en el vector peso han sido actualizados simultáneamente, lo que significa que no podemos volver a calcular el $\hat{y}^{(i)}$ antes de que todos los pesos Δw_j estén actualizados. Concretamente, para un conjunto de datos de dos dimensiones, podríamos escribir la actualización como:

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$

$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$

$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

Antes de implementar la regla del perceptrón en Python, haremos un sencillo experimento mental para mostrar la preciosa simplicidad de esta regla de aprendizaje. En los dos casos donde el perceptrón predice correctamente la etiqueta de clase, los pesos no cambian:

$$\Delta w_j = \eta\left(-1-(-1)\right)x_j^{(i)} = 0$$

$$\Delta w_j = \eta\left(1-1\right)x_j^{(i)} = 0$$

Sin embargo, en caso de una predicción errónea, los pesos se verán empujados hacia la dirección de la clase de destino negativa o positiva:

$$\Delta w_j = \eta\left(1--1\right)x_j^{(i)} = \eta\left(2\right)x_j^{(i)}$$

$$\Delta w_j = \eta\left(-1-1\right)x_j^{(i)} = \eta\left(-2\right)x_j^{(i)}$$

Para obtener una mejor intuición del factor multiplicativo $x_j^{(i)}$, veamos otro ejemplo sencillo, donde:

$$\hat{y}^{(i)} = -1, \quad y^{(i)} = +1, \quad \eta = 1$$

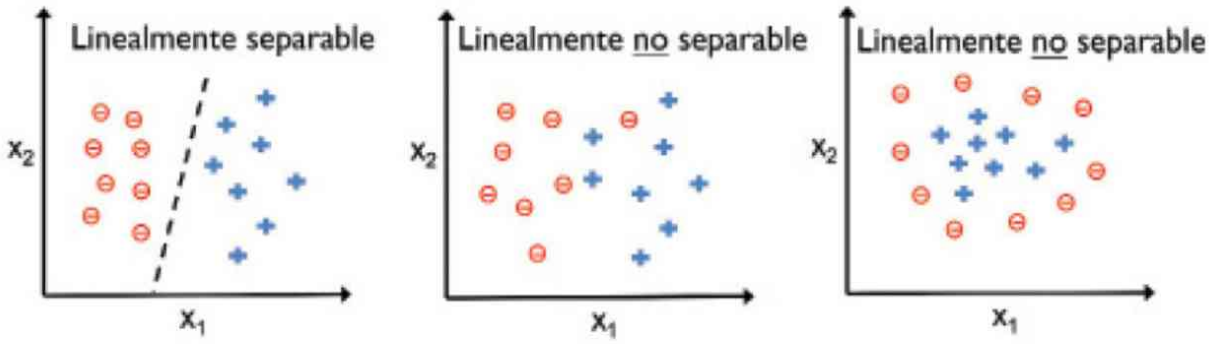
Asumimos que $x_j^{(i)} = 0.5$ y clasificamos erróneamente esta muestra como *-1*. En este caso, podríamos aumentar el correspondiente peso en 1, de manera que la entrada de red $x_j^{(i)} \times w_j$ fuera más positiva la próxima vez que se encontrar con esta muestra y, por tanto, fuera más probable que estuviera por encima del umbral de la función escalón unitario para clasificar la muestra como *+1*:

$$\Delta w_j = (1 - -1)0.5 = (2)0.5 = 1$$

La actualización del peso es proporcional al valor de $x_j^{(i)}$. Por ejemplo, si tenemos otra muestra $x_j^{(i)} = 2$ clasificada de forma incorrecta como *-1*, empujaríamos el límite de decisión por una medida aún mayor para clasificar la próxima vez correctamente esta muestra:

$$\Delta w_j = (1 - -1)2 = (2)2 = 4$$

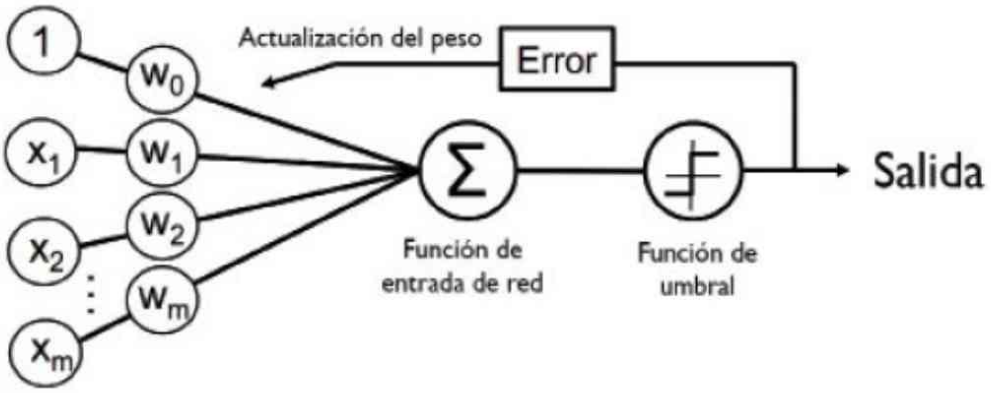
Es importante observar que la convergencia del perceptrón solo está garantizada si las dos clases son linealmente separables y si el rango de aprendizaje es suficientemente pequeño. Si las dos clases no pueden ser separadas por un límite de decisión lineal, podemos ajustar un número máximo de pasos sobre el conjunto de datos de entrenamiento **(épocas)** y/o un umbral para el número de clasificaciones erróneas toleradas. De otro modo, el perceptrón nunca dejaría de actualizar los pesos:



Descarga del código de ejemplo

Recuerda que en la parte inferior de la primera página del libro encontrarás el código de acceso que te permitirá descargar de forma gratuita los contenidos adicionales del libro.

Ahora, antes de saltar a la implementación en la sección siguiente, vamos a resumir cuanto acabamos de aprender en un simple diagrama que ilustra el concepto general del perceptrón:



Este diagrama muestra cómo el perceptrón recibe las entradas de una muestra **x** y las combina con los pesos **w** para calcular la entrada de red. A continuación, la entrada de red pasa por la función de umbral, que genera una salida binaria **-1** o **+1**; la etiqueta de clase predicha de la muestra. Durante la fase de aprendizaje, esta salida se utiliza para calcular el error de la predicción y actualizar los pesos.

Implementar un algoritmo de aprendizaje de perceptrón en Python

En la sección anterior, hemos aprendido cómo funciona la regla del perceptrón de Rosenblatt. Sigamos adelante. Vamos a implementarla en Python y a aplicarla al conjunto de datos Iris que presentamos en el *Capítulo 1, Dar a los ordenadores el poder de aprender de los datos.*

Una API perceptrón orientada a objetos

Vamos a tomar un enfoque orientado a objetos para definir la interfaz del perceptrón como una clase de Python, lo cual nos permite iniciar nuevos objetos **Perceptron** que pueden aprender de los datos mediante un método **fit** y hacer predicciones mediante un método **predict** separado. Como norma, agregamos un guion bajo (**_**) a aquellos atributos que no han sido creados durante la inicialización del objeto sino mediante la llamada de otros métodos del objeto, por ejemplo, **self.w_**.

Si aún no estás familiarizado con las librerías científicas de Python o necesitas un repaso, puedes consultar estos recursos (en inglés):

- NumPy: https://sebastianraschka.com/pdf/books/dlb/appendix_f_numpy-intro.pdf
- pandas: https://pandas.pydata.org/pandas-docs/stable/10min.html
- Matplotlib: http://matplotlib.org/users/beginner.html

Esta es la implementación de un perceptrón:

```
import numpy as np
class Perceptron(object):
  """Perceptron classifier.
  Parameters
  ------------
  eta : float
    Learning rate (between 0.0 and 1.0)
  n_iter : int
    Passes over the training dataset.
  random_state : int
    Random number generator seed for random weight
    initialization.
  Attributes
  -----------
  w_ : 1d-array
    Weights after fitting.
  errors_ : list
    Number of misclassifications (updates) in each epoch.
  """
  def __init__(self, eta=0.01, n_iter=50, random_state=1):
      self.eta = eta
```

```
        self.n_iter = n_iter
        self.random_state = random_state
    def fit(self, X, y):
        """Fit training data.
        Parameters
```

```
        ----------
        X : {array-like}, shape = [n_samples, n_features]
         Training vectors, where n_samples is the number of
         samples and
         n_features is the number of features.
        y : array-like, shape = [n_samples]
         Target values.
        Returns
        -------
        self : object
        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=1 + X.shape[1])
        self.errors_ = []
        for _ in range(self.n_iter):
          errors = 0
          for xi, target in zip(X, y):
           update = self.eta * (target - self.predict(xi))
           self.w_[1:] += update * xi
           self.w_[0] += update
           errors += int(update != 0.0)
          self.errors_.append(errors)
        return self
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def predict(self, X):
```

```
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Con esta implementación del perceptrón, ya podemos inicializar nuevos objetos **Perceptron** con un rango de aprendizaje proporcionado **eta** y **n_iter**, que es el número de épocas (pasos por el conjunto de entrenamiento). Mediante el método **fit**, inicializamos los pesos en **self.w_** para un vector $\mathbb{R}^{m+1}$, donde *m* significa el número de dimensiones (características) en el conjunto de datos, al cual añadimos *1* para el primer elemento en este vector que representa el parámetro del sesgo. Recuerda que el primer elemento en este vector, **self.w_[0]**, representa el denominado parámetro del sesgo del cual hemos hablado anteriormente.

Observa también que este vector contiene pequeños números aleatorios extraídos de una distribución normal con desviación estándar **0.01** con **rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])**, donde **rgen** es un generador de números aleatorios NumPy que hemos sembrado con una semilla aleatoria especificada por el usuario, por lo que podemos reproducir, si lo deseamos, resultados previos.

La razón por la cual no ponemos los pesos a cero es que el rango de aprendizaje η (**eta**) solo tiene efecto sobre el resultado de la clasificación si los pesos empiezan por valores distintos a cero. Si todos los pesos empiezan en cero, el parámetro **eta** del rango de aprendizaje afecta solo a la escala del vector peso, no a la dirección. Si estás familiarizado con la trigonometría, considera un vector $v1 = [1 \quad 2 \quad 3]$, donde el ángulo entre $v1$ y un vector $v2 = 0.5 \times v1$ sería exactamente cero, como queda demostrado en el siguiente fragmento de código:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
... np.linalg.norm(v2)))
0.0
```

En este caso, **np.arccos** es el coseno inverso trigonométrico y **np.linalg.norm** es una función que calcula la longitud de un vector. La razón por la cual hemos extraído los números aleatorios de una distribución normal aleatoria –en lugar de una distribución uniforme, por ejemplo– y por la que hemos utilizado una desviación estándar de **0.01** es arbitraria; recuerda que solo nos interesan valores pequeños aleatorios para evitar las propiedades de vectores todo cero, como hemos dicho anteriormente.

La indexación de NumPy para matrices unidimensionales funciona de forma

similar a las listas de Python, utilizando la notación de corchetes ([]). Para matrices bidimensionales, el primer indexador se refiere al número de fila y el segundo al número de columna. Por ejemplo, utilizaríamos **X[2, 3]** para seleccionar la tercera fila y la cuarta columna de una matriz X bidimensional **X**.

Tras haber puesto a cero los pesos, el método **fit** recorre todas las muestras individuales del conjunto de entrenamiento y actualiza los pesos según la regla de aprendizaje del perceptrón tratada en la sección anterior. Las etiquetas de clase son predichas por el método **predict**, que es llamado en el método **fit** para predecir la etiqueta de clase para la actualización del peso, aunque también puede ser utilizado para predecir las etiquetas de clase de nuevos datos una vez ajustado nuestro modelo. Además, también recopilamos el número de errores de clasificación durante cada época en la lista **self.errors_**, de manera que posteriormente podemos analizar si nuestro perceptrón ha funcionado bien durante el entrenamiento. La función **np.dot** que se utiliza en el método **net_input** simplemente calcula el producto escalar de un vector $w^T x$.

En lugar de utilizar NumPy para calcular el producto escalar entre dos matrices **a** y **b** mediante **a.dot(b)** o **np.dot(a, b)**, también podemos realizar el cálculo con Python puro mediante **sum([j * j for i, j in zip(a, b)])**. Sin embargo, la ventaja de utilizar NumPy frente a las estructuras clásicas de Python **for** loop es que sus operaciones

aritméticas son vectorizadas. La **vectorización** significa que una operación aritmética elemental se aplica automáticamente a todos los elementos de una matriz. Formulando nuestras operaciones aritméticas como una secuencia de instrucciones sobre una matriz, en lugar de llevar a cabo un conjunto de operaciones para cada elemento cada vez, se utilizan mejor las arquitecturas de CPU modernas con soporte SIMD (***Single Instruction, Multiple Data*** o, en español, Una Instrucción, Múltiples Datos). Además, NumPy utiliza librerías de álgebra lineal altamente optimizadas como la **Basic Linear Algebra Subprograms (BLAS)** y la **Linear Algebra Package (LAPACK)**, escritas en C o Fortran. Por último, NumPy también nos permite escribir nuestro código de un modo más compacto e intuitivo utilizando los conceptos básicos del álgebra lineal, como productos escalares de matrices y vectores.

Entrenar un modelo de perceptrón en el conjunto de datos Iris

Para probar nuestra implementación del perceptrón, vamos a cargar dos clases de flor, Setosa y Versicolor, del conjunto de datos Iris. Aunque la regla del perceptrón no está restringida a dos dimensiones, por razones de visualización solo tendremos en cuenta las características de longitud de sépalo y longitud de pétalo. Además, por razones prácticas, elegimos solo las dos clases de flor: Setosa y Versicolor. Sin embargo, el algoritmo perceptrón se puede ampliar a una clasificación multidimensional –por ejemplo, la técnica **One-versus-All (OvA)**–.

OvA, a veces también llamada **One-versus-Rest (OvR)**, es una técnica que nos permite ampliar un clasificador binario a problemas multiclase. Mediante OvA, podemos entrenar un clasificador por clase, donde cada clase individual se trata como una clase positiva y las muestras procedentes de otras clases se consideran clases negativas. Si tuviéramos que clasificar una nueva muestra de datos, utilizaríamos nuestros clasificadores *n*, donde *n* es el número de etiquetas de clase, y asignaríamos la etiqueta de clase con la fiabilidad más alta a cada muestra individual. En el caso del perceptrón, utilizaríamos OvA para elegir la etiqueta de clase asociada al mayor valor absoluto de entrada de red.

Primero, utilizaremos la librería **pandas** para cargar el conjunto de datos Iris

directamente del *UCI Machine Learning Repository* dentro de un objeto **DataFrame** e imprimir las últimas cinco líneas mediante el método **tail** para comprobar que los datos se han cargado correctamente:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
... 'machine-learning-databases/iris/iris.data',
... header=None)
>>> df.tail()
```

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Puedes encontrar una copia del conjunto de datos Iris (y de todos los otros conjuntos de datos utilizados en este libro) en el paquete de código de este libro, que puedes utilizar si estás trabajando *offline* o si el servidor UCI https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data está temporalmente no disponible. Por ejemplo, para cargar el conjunto de datos Iris desde el directorio local, puedes sustituir esta línea:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
'machine-learning-databases/iris/iris.data',
header=None)
```

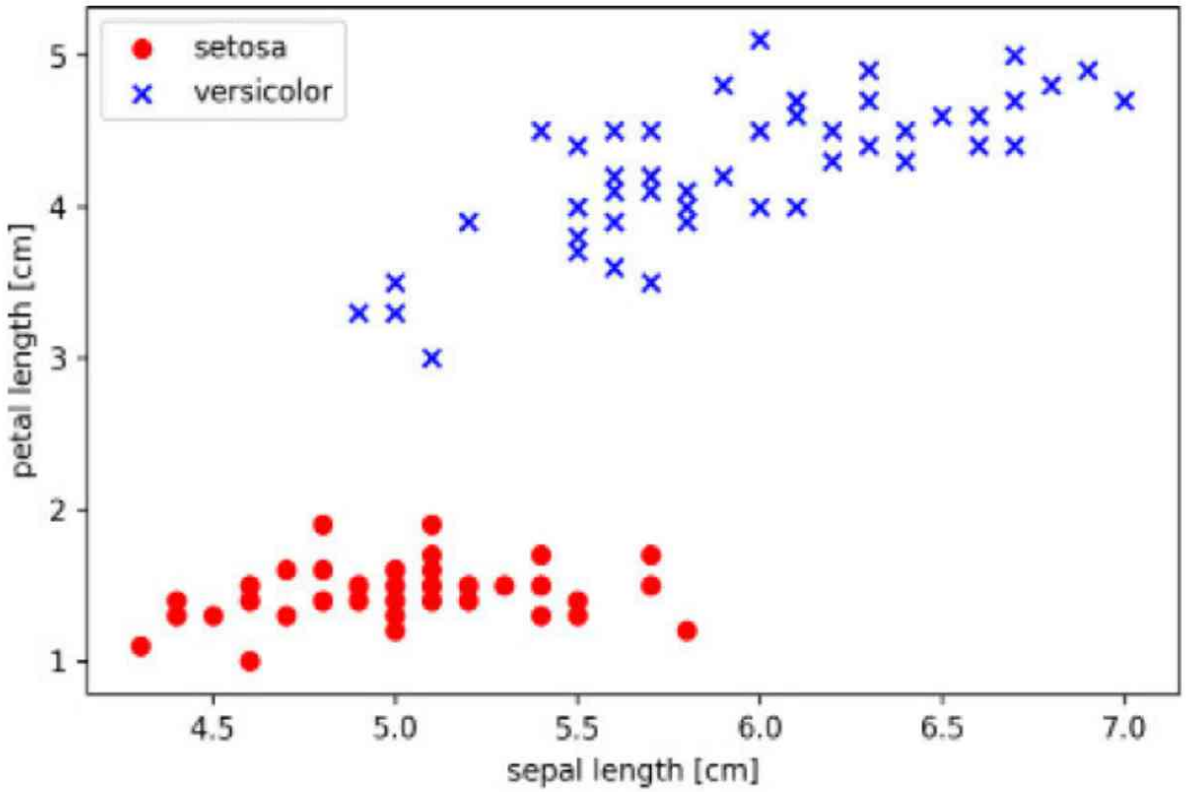
por esta otra:

```
df = pd.read_csv('your/local/path/to/iris.data',
header=None)
```

A continuación, extraemos las 100 primeras etiquetas de clase que corresponden a las 50 flores **Iris-setosa** y a las 50 **Iris-versicolor**, y convertimos las etiquetas de clase en las dos etiquetas de clase enteras **1** (**versicolor**) y **-1** (**setosa**) que asignamos a un vector **y**, donde el método de valores de un **DataFrame** pandas produce la correspondiente representación NumPy.
De forma similar, extraemos la primera columna de características (longitud del sépalo) y la tercera columna de características (longitud del pétalo) de las 100 muestras de entrenamiento y las asignamos a una matriz **X** de características, que podemos ver a través de un diagrama de dispersión bidimensional:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> # seleccionar setosa y versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> # extraer longitud de sépalo y longitud de pétalo
>>> X = df.iloc[0:100, [0, 2]].values
>>> # representar los datos
>>> plt.scatter(X[:50, 0], X[:50, 1],
... color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
... color='blue', marker='x', label='versicolor')
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Después de ejecutar el ejemplo de código precedente, deberíamos ver el siguiente diagrama de dispersión:



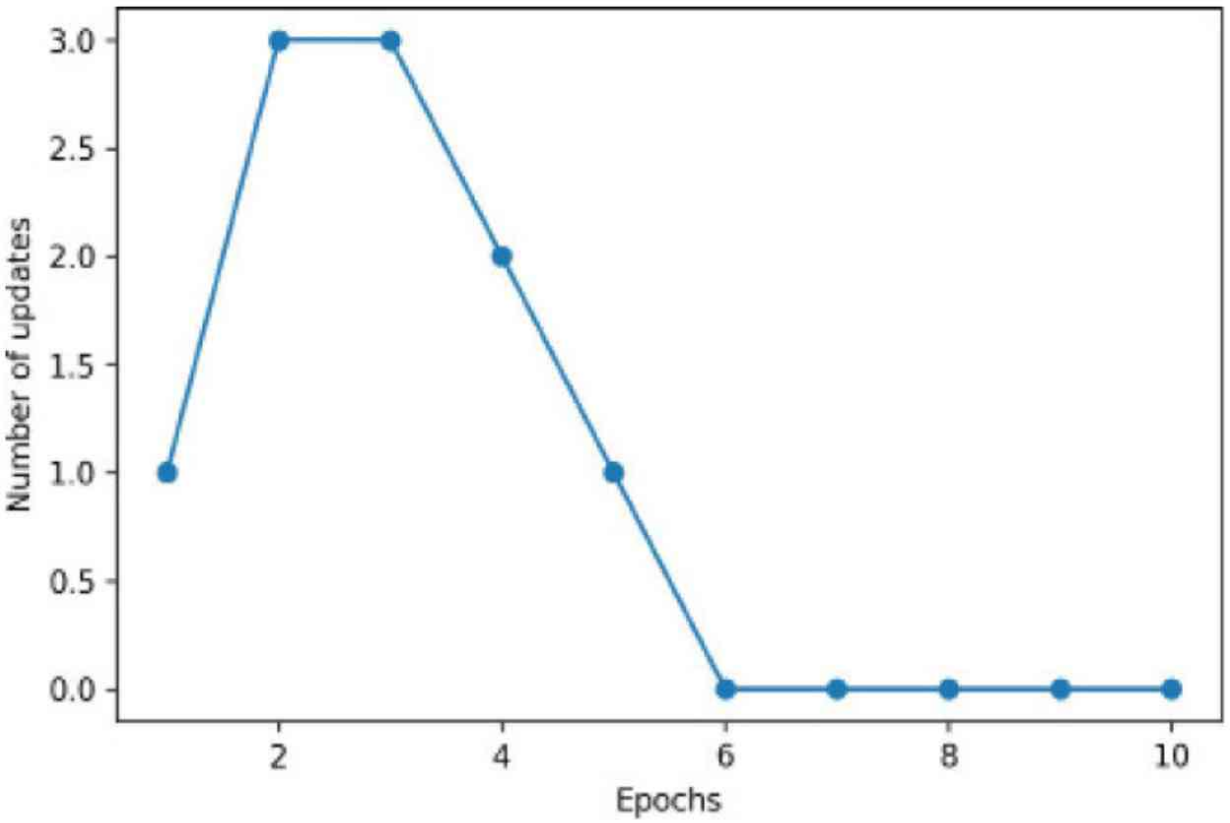
El diagrama de dispersión anterior muestra la distribución de las muestras de flor en el conjunto de datos Iris sobre los dos ejes de características, longitud del pétalo y longitud del sépalo. En este subespacio de características bidimensional, podemos ver que un límite de decisión lineal puede ser suficiente para separar flores Setosa de flores Versicolor. Por tanto, un clasificador lineal como el perceptrón podría ser capaz de clasificar las flores en este conjunto de datos perfectamente.

Ahora, ha llegado el momento de entrenar nuestro algoritmo de perceptrón en el subconjunto de datos Iris que acabamos de extraer. Además, reflejaremos en un gráfico el error de clasificación incorrecta para cada época para comprobar si el algoritmo ha convergido y encontrado un límite de decisión que separa las dos clases de flor Iris:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1),
... ppn.errors_, marker='o')
```

```
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of updates')
>>> plt.show()
```

Después de ejecutar el código anterior, deberíamos ver el diagrama de los errores de clasificación incorrecta frente al número de épocas, como se muestra a continuación:



Como podemos ver en el diagrama anterior, nuestro perceptrón ha convergido después de seis épocas y debería ser capaz de clasificar perfectamente las muestras de entrenamiento. Vamos a implementar una pequeña función de conveniencia para visualizar los límites de decisión para dos conjuntos de datos bidimensionales:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # definir un generador de marcadores y un mapa de colores
    markers = ('s', 'x', 'o', '^', 'v')
```

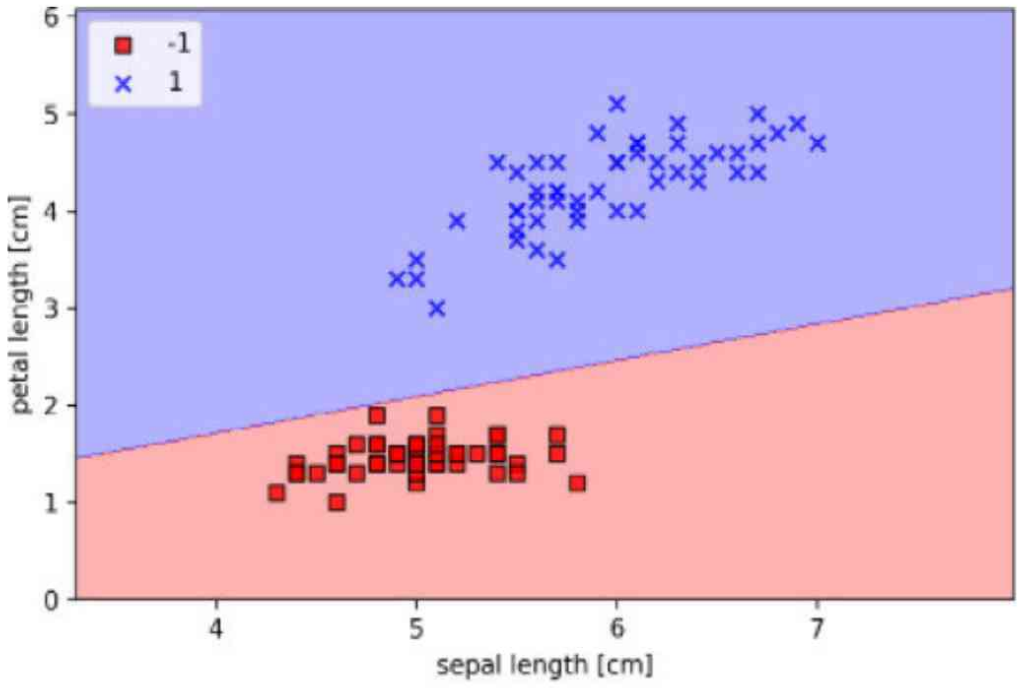
```
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    # representar la superficie de decisión
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    # representar muestras de clase
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=cl,
                    edgecolor='black')
```

En primer lugar, definimos un número de **colors** y **markers** y creamos un mapa de colores a partir de la lista de colores a través de **ListedColormap**. A continuación, determinamos los valores mínimos y máximos para las dos características y utilizamos los vectores de características para crear un par de matrices de cuadrícula **xx1** y **xx2** mediante la función **meshgrid** de NumPy. Como hemos entrenado nuestro clasificador de perceptrón en dos dimensiones de características, necesitamos acoplar las matrices y crear una matriz

que tenga el mismo número de columnas que el subconjunto de entrenamiento Iris. Para ello, podemos utilizar el método **predict** para predecir las etiquetas de clase **Z** de los correspondientes puntos de la cuadrícula. Después de remodelar las etiquetas de clase **Z** predichas en una cuadrícula con las mismas dimensiones que **xx1** y **xx2**, ya podemos dibujar un diagrama de contorno con la función **contourf** de Matplotlib, que mapea las diferentes regiones de decisión en distintos colores para cada clase predicha en la matriz de cuadrícula:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Después de ejecutar el ejemplo de código anterior, deberíamos ver un diagrama de las regiones de decisión, como se muestra en la siguiente figura:



Como podemos ver en el diagrama, el perceptrón ha aprendido un límite de decisión capaz de clasificar perfectamente todas las muestras de flor en el

subconjunto de entrenamiento Iris.

Aunque el perceptrón ha clasificado a la perfección las dos clases de flor Iris, la convergencia es uno de los mayores problemas del perceptrón. Frank Rosenblatt probó matemáticamente que la regla de aprendizaje del perceptrón converge si las dos clases pueden ser separadas por un hiperplano lineal. Sin embargo, si las clases no pueden ser separadas perfectamente por un límite de decisión lineal, los pesos no dejarán nunca de actualizarse a menos que indiquemos un número máximo de épocas.

Neuronas lineales adaptativas y la convergencia del aprendizaje

En esta sección, echaremos un vistazo a otro tipo de red neuronal de capa única: las neuronas lineales adaptativas, en inglés **ADAptive LInear NEuron (Adaline)**. Adaline fue publicada por Bernard Widrow y su alumno Ted Hoff, pocos años después del algoritmo de perceptrón de Frank Rosenblatt, y puede considerarse como una mejora de este último. (Puedes consultar *An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2, B. Widrow and others, Stanford Electron Labs*, Stanford, CA, *October 1960*).

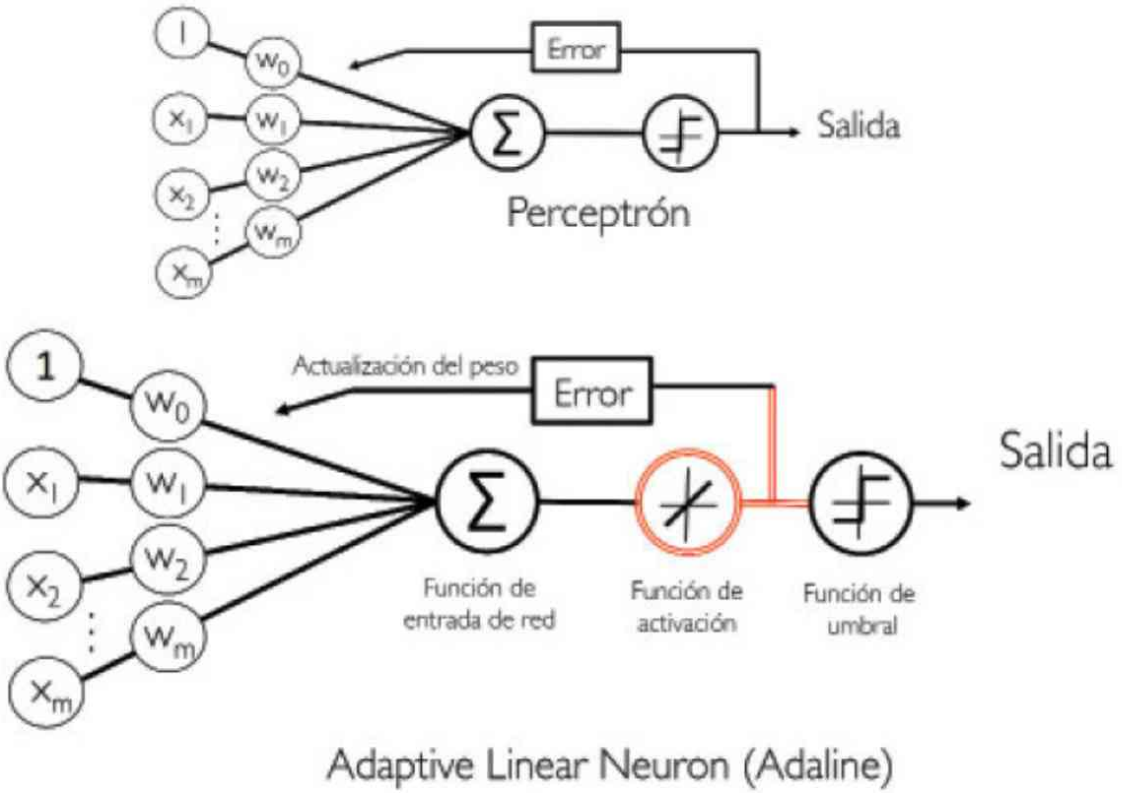
El algoritmo Adaline es especialmente interesante porque ilustra los conceptos clave para definir y minimizar las funciones de coste continuas. Esto sienta las bases para la comprensión de algoritmos de aprendizaje automático más avanzados para la clasificación, como la regresión logística, máquinas de vectores de soporte y modelos de regresión, que trataremos en capítulos posteriores.

La diferencia clave entre la regla Adaline (también conocida como *regla Widrow-Hoff*) y el perceptrón de Rosenblatt es que los pesos se actualizan en base a una función de activación lineal en vez de en base a una función escalón unitario como sucede en el perceptrón. En Adaline, esta función de activación lineal $\phi(z)$ es simplemente la función de identificación de la entrada de red, por lo que:

$$\phi\left(\boldsymbol{w}^T \boldsymbol{x}\right) = \boldsymbol{w}^T \boldsymbol{x}$$

Mientras que la función de activación lineal se utiliza para aprender los pesos, seguimos utilizando una función de umbral que realiza la predicción final, que es parecida a la función escalón unitario que hemos visto anteriormente. Las diferencias principales entre el perceptrón y el algoritmo

Adaline se encuentran destacadas en la siguiente imagen:



Esta ilustración muestra que el algoritmo Adaline compara las etiquetas de clase verdaderas con la salida de valores continuos de la función de activación lineal para calcular el error del modelo y actualizar los pesos. Por el contrario, el perceptrón compara las etiquetas de clase verdaderas con las etiquetas de clase predichas.

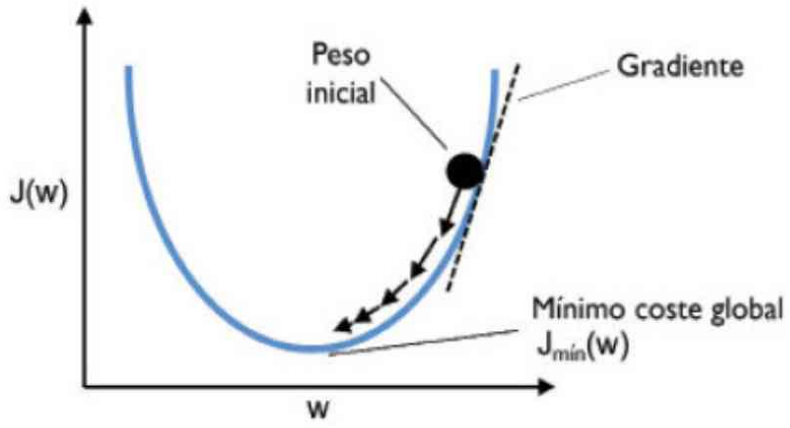
Minimizar funciones de coste con el descenso de gradiente

Uno de los ingredientes clave de los algoritmos de aprendizaje automático supervisado es una **función objetivo** definida que debe ser optimizada durante el proceso de aprendizaje. Esta función objetivo suele ser una función de coste que queremos minimizar. En el caso de Adaline, podemos definir la función de coste J para aprender los pesos como la **Suma de Errores Cuadráticos** (**SSE**, del inglés *Sum of Squared Errors*) entre la salida calculada y la etiqueta de clase verdadera:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

Hemos añadido el término $\frac{1}{2}$ simplemente para facilidad nuestra, porque nos permitirá derivar el gradiente de un modo más fácil, como veremos en los siguientes párrafos. La principal ventaja de esta función de activación lineal continua, en comparación con la función escalón unitario, es que la función de coste pasa a ser diferenciable. Otra propiedad a tener en cuenta de esta función de coste es que es convexa; esto significa que podemos utilizar un simple pero potente algoritmo de optimización, denominado **descenso de gradiente**, para encontrar los pesos que minimicen nuestra función de coste y clasificar así las muestras del conjunto de datos Iris.

Como se muestra en la imagen siguiente, podemos describir la idea principal que hay detrás del descenso de gradiente como *bajar una colina* hasta obtener un mínimo de coste global o local. En cada iteración, realizamos un paso en la dirección opuesta del gradiente donde el tamaño del paso está determinado por el valor del rango de aprendizaje, así como por la pendiente del gradiente:



Con el descenso de gradiente, podemos actualizar los pesos haciendo un paso en la dirección opuesta del gradiente $\nabla J(\boldsymbol{w})$ de nuestra función de coste $J(\boldsymbol{w})$:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

donde el cambio de peso $\Delta \boldsymbol{w}$ se define como el gradiente negativo multiplicado por el rango de aprendizaje η :

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w})$$

Para calcular el gradiente de la función de coste, necesitamos calcular la derivación parcial de la función de coste con respecto a cada peso w_j :

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Por lo que podemos escribir la actualización del peso w_j como:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Como actualizamos todos los pesos a la vez, nuestra regla de aprendizaje Adaline es:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

Para los que están familiarizados con

el cálculo, la derivación parcial de la función de coste SSE con respecto al peso *j* se puede obtener así:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2}\sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \sum_k \left(w_k x_k^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\left(-x_j^{(i)}\right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}$$

Aunque la regla de aprendizaje Adaline puede parecer idéntica a la regla del perceptrón, podemos observar que el $\phi\left(z^{(i)}\right)$ con $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$ es un número real y una etiqueta de clase completa. Además, la actualización del peso se calcula en base a todas las muestras del conjunto de entrenamiento (en lugar de actualizar los pesos de forma incremental después de cada muestra), razón por la cual este enfoque también se conoce como **descenso de gradiente en lotes**.

Implementar Adaline en Python

Como la regla del perceptrón y Adaline son muy parecidos, tomaremos la implementación del perceptrón que definimos anteriormente y cambiaremos el método **fit** de manera que los pesos se actualicen minimizando la función de coste mediante el descenso de gradiente:

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.
    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.
    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    cost_ : list
      Sum-of-squares cost function value in each epoch.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
    def fit(self, X, y):
```

```
        """ Fit training data.
        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
         Training vectors, where n_samples is the number of
         samples and
         n_features is the number of features.
        y : array-like, shape = [n_samples]
         Target values.
        Returns
        -------
        self : object
        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=1 + X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
          net_input = self.net_input(X)
          output = self.activation(net_input)
          errors = (y - output)
          self.w_[1:] += self.eta * X.T.dot(errors)
          self.w_[0] += self.eta * errors.sum()
          cost = (errors**2).sum() / 2.0
          self.cost_.append(cost)
        return self
    def net_input(self, X):
        """Calculate net input"""
```

```
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def activation(self, X):
        """Compute linear activation"""
        return X
    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X))
                >= 0.0, 1, -1)
```

En lugar de actualizar los pesos después de evaluar cada muestra de entrenamiento individual, como en el perceptrón, calculamos el gradiente en base a todo el conjunto de datos de entrenamiento mediante **self.eta * errors.sum()** para el parámetro del sesgo (peso cero) y mediante **self.eta * X.T.dot(errors)** para los pesos 1 a *m*, donde **X.T.dot(errors)** es una multiplicación matriz por vector entre nuestra matriz de características y el vector de error.