Aprendizaje supervisado: problemas de regresión y combinación de métodos

PID_00279451

Raúl Benítez
Andrés Cencerrado
Gerard Escudero
Lluís Gómez
Samir Kanaan
Félix José Fuentes
Marc Maceira
David Masip
Vicenç Torra
Carles Ventura

Tiempo mínimo de dedicación recomendado: 2 horas

Universitat Oberta de Catalunya

Raúl Benítez | **Andrés Cencerrado** | **Gerard Escudero**

Lluís Gómez | **Samir Kanaan** | **Félix José Fuentes**

Marc Maceira | **David Masip** | **Vicenç Torra**

Carles Ventura

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por los profesores: Marc Maceira Duch, Carles Ventura Roy

Primera edición: febrero 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Raúl Benítez, Andrés Cencerrado, Gerard Escudero, Lluís Gómez, Samir Kanaan, Félix José Fuentes, Marc Maceira, David Masip, Vicenç Torra, Carles Ventura
Producción: FUOC

Índice

Introducción

En este módulo se estudiarán dos temáticas muy diferenciadas: los modelos de regresión y la combinación de métodos. Cada bloque temático se desarrolla en un apartado diferente.

El apartado 1 define las diferentes técnicas de regresión. Si recordamos lo que vimos en el módulo «Introducción al aprendizaje automático», los problemas de regresión se definen como escenarios de aprendizaje automático supervisado (en el que cada instancia o ejemplo incluye un atributo con la solución) en el que el atributo es de tipo numérico. Como ejemplo de problema de regresión, vimos el caso en el que queríamos crear un modelo capaz de predecir el precio de una casa a partir de los metros cuadrados. Así, la entrada a nuestro modelo consiste en una serie de valores numéricos (como por ejemplo, los metros cuadrados de la casa, el número de habitaciones...) y la salida es también un valor numérico (como por ejemplo, el precio). Algunos de los algoritmos presentados en el apartado 1 parten del módulo anterior «Problemas de clasificación» y cambian el planteamiento para tratar el problema de la regresión.

El apartado 2 se ocupa de la combinación de los métodos. Esta combinación es posible tanto con los problemas de clasificación estudiados en el módulo «Aprendizaje supervisado: problemas de clasificación» como con los problemas de regresión del apartado 1 de este módulo. En los casos de uso reales, una vez que hemos implementado y probado los diferentes métodos, nos encontramos que cada tipología de métodos o cada realización obtiene resultados diferentes. Para analizar esta casuística, podemos recordar el problema del sesgo y la varianza estudiados en el módulo 1. La generalización a partir de un conjunto de ejemplos genera un compromiso entre el sesgo y la varianza. Si aumentamos la complejidad del modelo, el sesgo baja, pero la varianza aumenta. Para mejorar este compromiso, hay técnicas que permiten reducir la varianza a pesar de usar modelos complejos, como la combinación de métodos. Mediante esta combinación de métodos, obtendremos soluciones con mejores relaciones entre el sesgo y la varianza que si utilizamos modelos simples.

1. Modelos de regresión

En este apartado partiremos de la definición de un clasificador para plantear el problema de la regresión. Tal como hemos visto en el módulo didáctico anterior, dado un vector de entrada $\mathbf{x}$, un clasificador produce como salida una etiqueta o identificador de clase $y \in \{0,\dots,C-1\}$; en el que C es el número de clase. Este número de clase C es un número entero positivo. La regresión es otra tarea muy común en el aprendizaje computacional en el que la salida es ahora un número real, de forma que se trata del análogo continuo de un clasificador. Por lo tanto, aprender un modelo de regresión con un conjunto de entrenamiento:

$$\{(x_1,y_1),(x_2,y_2),\dots,(x_m,y_m)\}, \text{ con } x_i \in \mathbb{R}^n, \text{ y } y_i \in \mathbb{R}$$

significa aprender una función escalar $h(x)$ de n variables. Los algoritmos que estudiaremos en este apartado obtendrán funciones $h(x)$ que permitirán inferir valores reales $\hat{y}_i$ dada una entrada x_i. En el ejemplo del precio de las casas, $\hat{y}_i$ correspondería al precio predicho de la casa, mientras que x_i correspondería a las características de la casa (metros cuadrados, número de habitaciones...).

1.1 Regresión lineal

El modelo de regresión lineal ya se ha visto en asignaturas previas. En este módulo didáctico lo veremos de manera resumida a modo de recordatorio y nos servirá para introducir otros modelos de regresión que veremos más adelante.

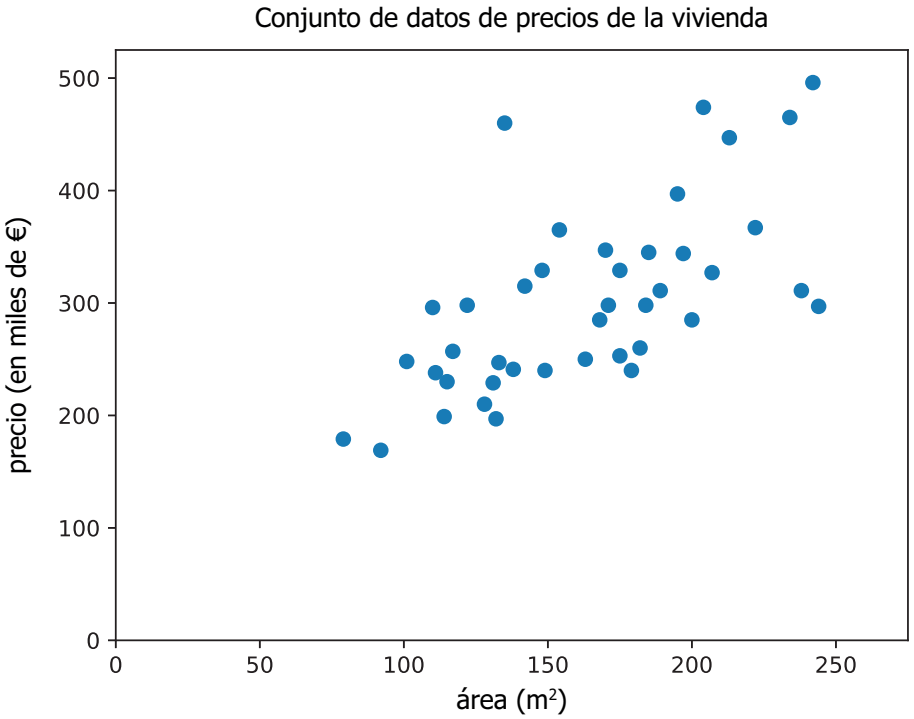
Un modelo de regresión lineal se utiliza para explicar una variable determinada y, en función de una variable escalar x o bien, en función de varias variables $(x_1,x_2,\dots,x_n)$ que representaremos en forma de vector, como x. En el primer caso, cuando la entrada es una única variable escalar $x \in \mathbb{R}$, hablaremos de regresión univariante y, en el segundo caso, cuando la entrada es $x \in \mathbb{R}^n$, de regresión multivariante. Como su nombre indica, el modelo que utilizaremos en ambos casos es una función lineal de las variables de entrada.

1.1.1 Regresión lineal univariante

Para estudiar el modelo de regresión lineal univariante, recuperamos un ejemplo que ya vimos en el módulo didáctico «Introducción al aprendizaje auto-

mático», cuando introdujimos la tarea de regresión. El ejemplo consiste en un modelo para predecir el precio de una casa a partir de los metros cuadrados. En la figura 1, se muestra un conjunto de datos donde cada punto representa un par de entrada y de salida en este modelo.

Figura 1. Ejemplo de conjunto de datos para un problema de regresión



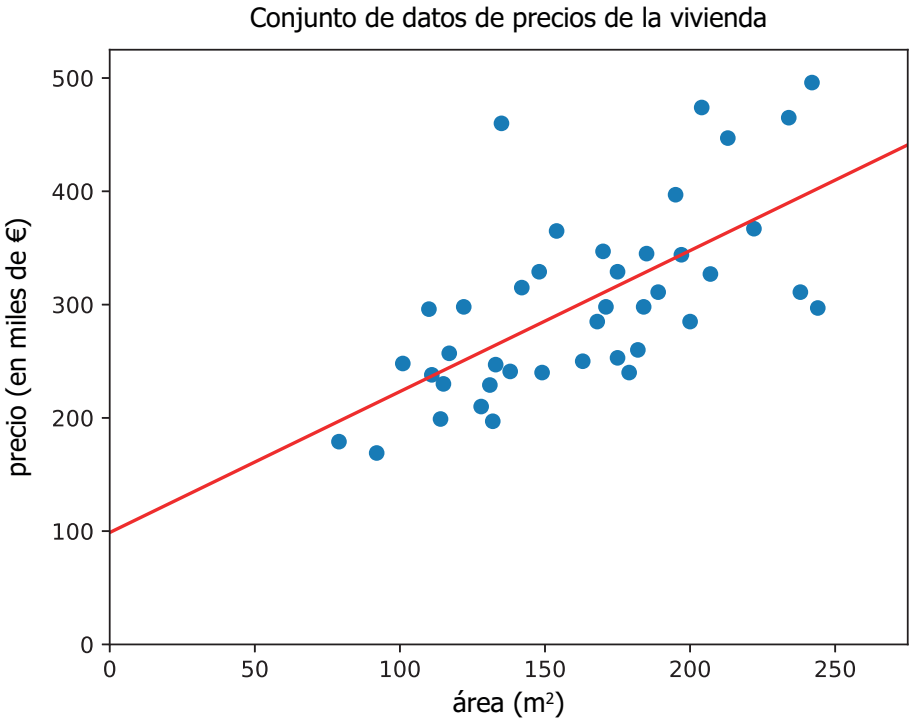
Recordad que hablamos de un problema de aprendizaje supervisado, puesto que disponemos del valor de salida correcto (y) para cada uno de los ejemplos del conjunto de datos de entrenamiento.

La salida de un modelo de regresión lineal es una función lineal de la variable de entrada:

$$h(x) = \theta_1 x + \theta_0$$

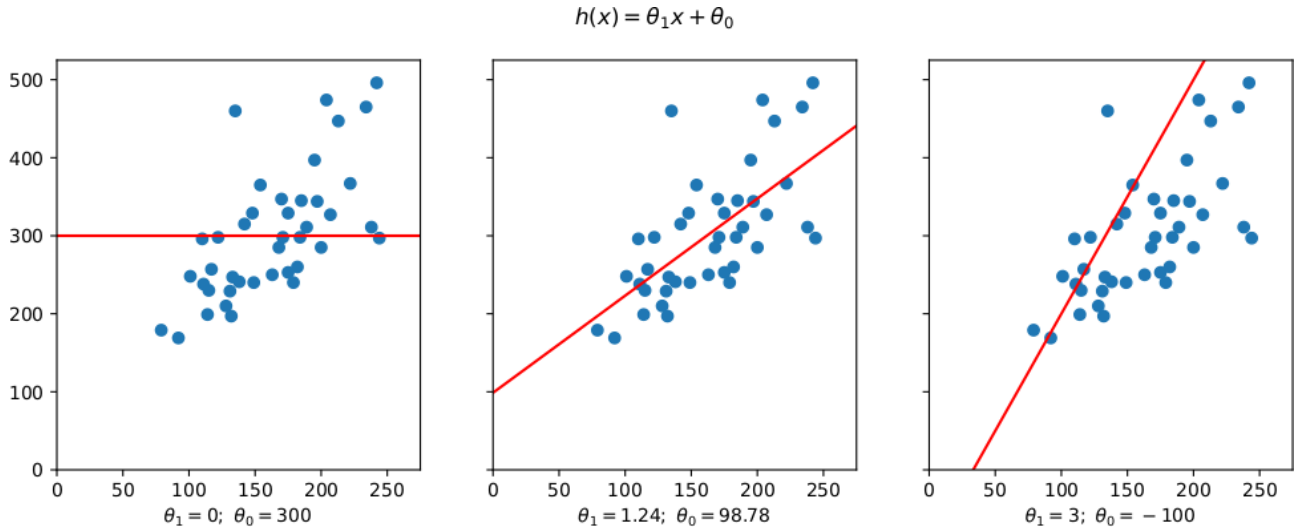
donde θ_1 y θ_0 son los parámetros del modelo, que controlan, respectivamente, la pendiente y el punto de corte de la recta $h(x)$ con el eje vertical, tal como ilustra la figura 2. Como ya hemos dicho anteriormente, denominamos a este modelo regresión lineal simple con una variable o regresión lineal univariante.

Figura 2. Ejemplo de regresión lineal. La función $h(x)$ se ajusta a los ejemplos de nuestro conjunto de datos de entrenamiento y nos permite hacer predicciones para la variable de salida en función de la variable de entrada.



Una vez que hemos escogido la función $h(x)$, en este caso lineal, para nuestro modelo, tenemos que ver como se eligen los valores de los parámetros de esta función (θ_1 y θ_0) de forma que se ajuste lo mejor posible a nuestro conjunto de datos de entrenamiento. En la figura 3 se muestran varios modelos de regresión lineal para diferentes valores de los parámetros θ_1 y θ_0.

Figura 3. Ejemplos de modelos de regresión lineal para diferentes valores de y θ_1 i θ_0



En el marco del aprendizaje automático, la intuición es que tenemos que escoger los valores de θ_1 y θ_0 de forma que la respuesta del modelo ($h(x)$) para cada posible entrada x de nuestro conjunto de entrenamiento sea lo más cercana posible a su valor y correspondiente. Podemos formalizar esta intuición de la manera siguiente:

$$\theta_0,\theta_1 = \underset{\theta_0,\theta_1}{\operatorname{argmin}} \sum_{i=0}^{m} (h(x_i) - y_i)^2$$

en el que m es el número de ejemplos en el conjunto de datos de entrenamiento. Es decir, la función que queremos minimizar para optimizar el modelo de regresión lineal no es otra que el error cuadrático medio.* El error cuadrático medio es la función de coste utilizada habitualmente para resolver los problemas de regresión. Por convención, definimos la función de coste J para el modelo de regresión como:

* En inglés, *mean squared error* (MSE).

Ved también

Ved el apartado «Métricas de evaluación» del módulo didáctico «Introducción al aprendizaje automático», en el que se introdujo el error cuadrático medio como métrica de evaluación para modelos de regresión.

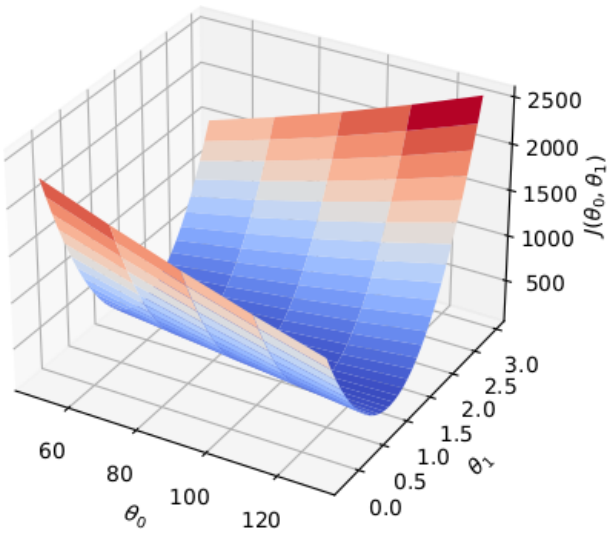
$$J(\theta_0,\theta_1) = \frac{1}{2m} \sum_{i=0}^{m} (h(x_i) - y_i)^2$$

y lo que queremos es encontrar los valores de los parámetros que minimicen esta función de coste:

$$\theta_0,\theta_1 = \underset{\theta_0,\theta_1}{\operatorname{argmin}} J(\theta_0,\theta_1)$$

Este es un problema de optimización de funciones que, como ya sabéis, podemos resolver de maneras diferentes. En el contexto del aprendizaje automático, lo más común es utilizar el método del descenso de gradientes. Para este problema, la función de coste J está muy definida y es continua en cualquier punto del espacio de búsqueda de los parámetros. Por lo tanto, es una función derivable y el valor de la derivada en un punto nos permitirá guiar la búsqueda de los parámetros hacia el valor mínimo de J. Además, se da el caso en que la función de coste MSE es una función convexa para el caso de la regresión lineal univariante y, por lo tanto, tiene un único valor mínimo absoluto. La figura 4 muestra la función de coste para los datos de los precios de la vivienda de la figura 1.

Figura 4. La función de coste J está muy definida y es continua en cualquier punto del espacio de búsqueda de los parámetros θ_1 y width=8cm θ_0.



Omitiremos en este módulo los detalles del método de descenso de gradientes, entendiendo que ya se han visto en asignaturas previas y que se volverán a ver más adelante en el módulo dedicado a redes neuronales en el contexto del entrenamiento de las redes neuronales.

1.1.2 Regresión lineal multivariante

En el ejemplo que hemos usado hasta ahora para el modelo de regresión lineal univariante, se podía explicar la variable de salida (el precio de una casa) a partir de una única variable de entrada (los metros cuadrados). Ahora ampliaremos este ejemplo para el caso multivariante en el que tendremos más de una variable de entrada en forma de diferentes características de la casa, a partir de las que querremos hacer predicciones de la variable de salida. La tabla 1 muestra algunos ejemplos de un posible conjunto de datos de entrenamiento para este caso.

Tabla 1. Ejemplos de un conjunto de datos para un modelo de regresión lineal multivariante

m^2	# habitaciones	# pisos	Año de construcción	Precio (en miles de euros)
90	3	1	2010	210
79	2	1	2012	179
120	4	2	2015	264
92	2	1	1995	150
...	...	...	...	...

Para cada ejemplo del conjunto de datos de entrenamiento, denotaremos como $x \in \mathbb{R}^n$ la variable de entrada, un vector con las n características ($n = 4$ a la tabla 1) y continuaremos utilizando y para la variable de salida. Para referirnos al y-ésimo ejemplo en el conjunto de datos de entrenamiento, utilizaremos la notación (x_i, y_i), mientras que, para referirnos a cada una de las varias características de este ejemplo de manera individual, utilizaremos $(x_{i,1}, x_{i,2}, ..., x_{i,n})$.

Con esta notación, el modelo que utilizaremos para la regresión lineal multivariante toma la forma siguiente:

$$h(x) = \theta_n x_n + \cdots + \theta_2 x_2 + \theta_1 x_1 + \theta_0$$

en el que $\theta = \{\theta_0, \theta_1, \ldots, \theta_n\}$ son los parámetros de un hiperplano en el espacio $\mathbb{R}^n$. Por conveniencia, añadiremos una nueva característica x_0 con un valor igual a 1 en todos los ejemplos, de forma que podremos escribir nuestro modelo de la forma más compacta, como el producto escalar* de los vectores $\theta^T \in \mathbb{R}^{n+1}$ y $x \in \mathbb{R}^{n+1}$:

*** En inglés, *inner product*, o también *dot product*.**

$$h(x) = \theta_n x_n + \cdots + \theta_2 x_2 + \theta_1 x_1 + \theta_0 x_0;\ x_0 = 1$$

$$h(x) = \theta^T x$$

La función de coste tomará ahora la forma siguiente:

$$J(\theta_0,\theta_1,\ldots,\theta_n) = \frac{1}{2m}\sum_{i=0}^{m}(\theta^T x^{(i)} - y^{(i)})^2$$

De igual manera que en el caso univariante, la función de coste J es derivable y convexa. Por lo tanto, también en el caso de la multivariante podremos utilizar el método del descenso de gradientes para encontrar los valores óptimos de los parámetros θ que minimizan la función de coste en nuestro conjunto de datos de entrenamiento.

1.2 kNN

En el módulo didáctico 2 («Aprendizaje supervisado: problemas de clasificación») hemos introducido el algoritmo kNN (k - vecino más próximo)* como método de clasificación basado en el cálculo de distancias. Veremos ahora como el algoritmo kNN se puede utilizar en el caso de que la variable para predecir sea continua, en lugar de discreta, como en el caso de la clasificación.

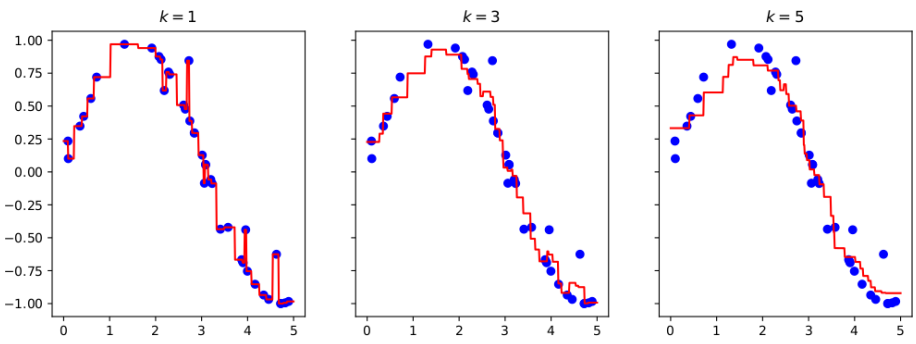
*** En inglés, *k nearest neighbors.***

Una posible implementación muy sencilla para un modelo de regresión kNN consiste simplemente en calcular la media del valor de salida y de los k vecinos más próximos:

$$y = \frac{1}{k}\sum_{i\in kNN(x)} y_i$$

en el que $kNN(x)$ es el conjunto de los k vecinos más próximos dadas una entrada x y una función de distancia determinadas. La figura 5 muestra tres ejemplos de esta implementación para los diferentes valores de k, en la que los puntos representan los pares de entrada y salida de nuestro conjunto de datos $\{(x_1,y_1),(x_2,y_2),\ldots,(x_m,y_m)\}$ y la curva roja es la predicción y del modelo para cualquier valor de entrada x.

Figura 5. Ejemplos de regresión kNN para los diferentes valores de width=12cm k



Fijaos en que el modelo de regresión kNN se adapta a los conjuntos de datos no lineales de manera natural y que puede aproximar funciones tanto de una sola variable de entrada (como en la figura 5), como de varias variables de en-

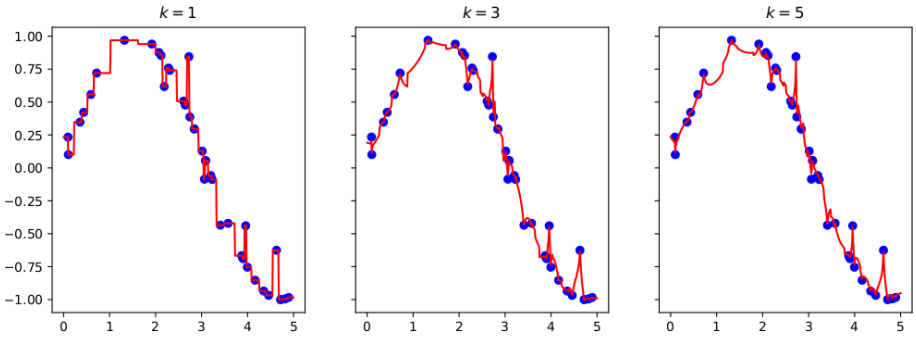
trada $(x_1, x_2, \ldots, x_n)$. Por otro lado, hay que tener en cuenta que, como también pasa en el caso de la clasificación, el rendimiento del modelo dependerá de la densidad de los ejemplos de los que disponemos en una región determinada del espacio de posibles entradas.

Una variación de este modelo de regresión kNN utiliza una media ponderada del valor de salida y de los k vecinos más próximos:

$$y = \frac{1}{k} \sum_{i \in kNN(x)} w_i y_i$$

donde w_i es un valor entre 0 y 1 que se calcula como función inversa de la distancia de cada vecino en el punto de entrada x. De este modo, los puntos más próximos a la entrada x contribuyen más a la regresión que los puntos lejanos. La figura 6 muestra tres ejemplos de esta implementación para los diferentes valores de k. Para $k = 1$, el modelo es equivalente al de la figura 5.

Figura 6. Ejemplos de regresión kNN con media ponderada por la inversa de la distancia para los diferentes valores de k



Finalmente, merece la pena resaltar en este punto que todas las consideraciones que hicimos en el módulo 2 sobre el algoritmo kNN para la clasificación –por ejemplo, en cuanto a la eficiencia computacional, la elección de la medida de la distancia y del valor de k, etc.– continúan teniendo la misma importancia en el caso de la tarea de regresión.

1.3 Árboles de decisión

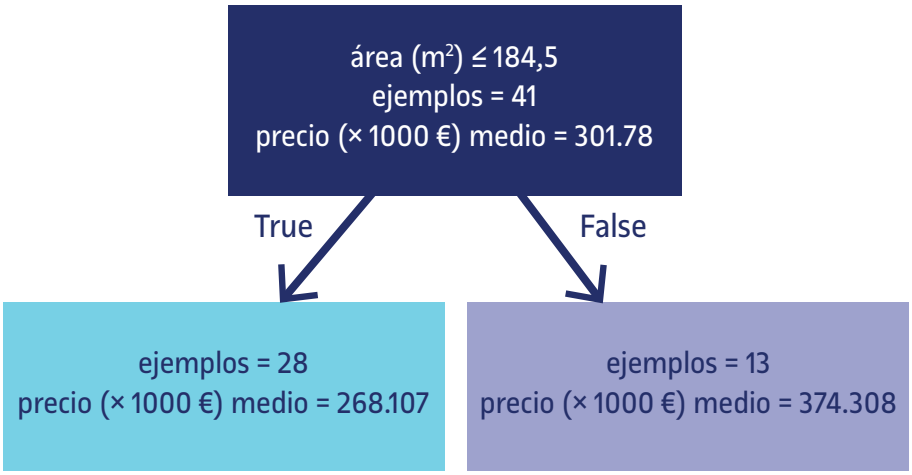
Los árboles de decisión que hemos visto en el módulo 2 para la clasificación también se pueden adaptar con una ligera modificación para hacer la regresión. El proceso iterativo de construcción del árbol (aprendizaje) y el uso posterior para hacer predicciones son casi idénticos en ambos casos, pero en los árboles de decisión para la regresión, la medida de bondad de las particiones y la manera de asignar clases a los nodos terminales (hojas) del árbol cambian.

En los nodos terminales sustituiremos la etiqueta del nodo (identificador de clase, en el caso de la clasificación) por un número real que sintetice las muestras que le llegan durante el entrenamiento. Por ejemplo, podemos utilizar la

media de sus valores de salida y. De este modo, al hacer la inferencia para una entrada x del conjunto de datos de prueba, le asignaremos como predicción y el valor asociado con el nodo terminal al que llegamos después de recorrer el árbol de decisión.

La figura 7 muestra un árbol de decisión con un único nodo de decisión y dos nodos terminales para el conjunto de datos de los precios de la vivienda. La figura 8 muestra la función de regresión ($h(x)$) de este árbol. Fijaos en que asignar la media de los valores y de un subconjunto de los datos de entrenamiento como etiqueta de los nodos terminales es equivalente a hacer la regresión lineal con una recta con pendiente 0 para este subconjunto. Además, podemos ver que, si utilizamos los árboles de decisión, estamos convirtiendo de alguna manera un problema de regresión en uno de clasificación –fijaos que en la figura 8, el algoritmo ha encontrado un punto de corte en el espacio de la variable de entrada para dividir las casas en económicas o caras (dos clases) en función de su área–, con la diferencia de que la etiqueta de cada clase es un número real (el precio medio de cada subconjunto). Obviamente, lo interesante del algoritmo es que podemos continuar haciendo tantas particiones de estos subconjuntos como queramos.

Figura 7. Árbol de decisión para la regresión con un único nodo de decisión y dos nodos terminales, para el conjunto de datos de los precios de la vivienda

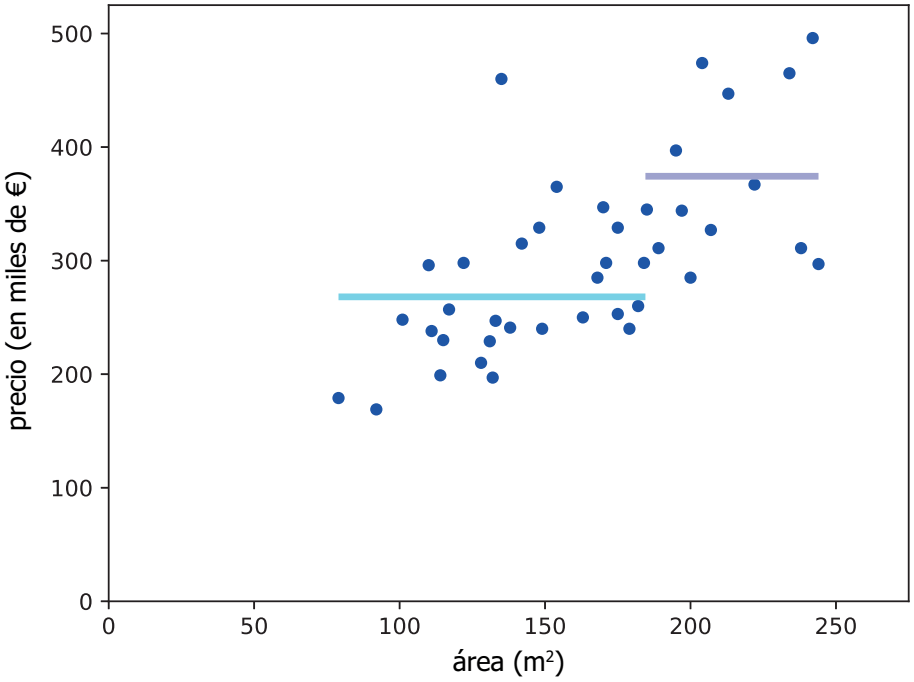


Como hemos mencionado anteriormente, el criterio de división de los datos durante la construcción del árbol también cambia en el caso de la regresión. La intuición consiste en que tenemos que encontrar la partición (punto de corte) mediante la que, en los dos subconjuntos resultantes, se minimice el error de regresión utilizando una recta con pendiente 0. Dicho de otro modo, buscaremos la partición que minimice la suma de las varianzas de y de los subconjuntos izquierdo y derecho:

$$\underset{A,B}{\operatorname{argmin}} \left(Var_y(A) + Var_y(B)\right)$$

donde A y son B los subconjuntos izquierdo y derecho, respectivamente, y es $Var_y(\cdot)$ la varianza de los valores y del subconjunto de datos dado. Fijaos que la

Figura 8. Función de regresión del árbol de decisión de la figura 7. Los colores de las dos rectas de regresión indican los nodos terminales correspondientes.



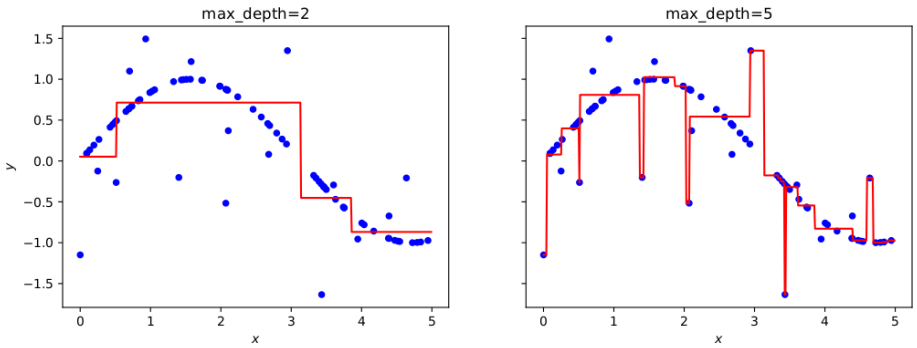
varianza equivale al cálculo del error cuadrático medio (MSE) por la recta con pendiente 0 que corta el eje vertical en el valor de la media de los datos (μ):

$$Var(Y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mu)^2$$

En la figura 9 se utilizan dos árboles de decisión para adaptarse a un conjunto de datos dado con dos valores diferentes para el nivel de profundidad máxima del árbol. Podemos ver que, si la profundidad máxima del árbol es demasiado grande, los árboles de decisión pueden aprender detalles demasiado específicos de los datos de entrenamiento, sobreespecializándose* y no generalizando en nuevos datos de prueba. Además, los árboles de decisión pueden ser inestables porque las pequeñas variaciones en el conjunto de datos de entrenamiento pueden generar un árbol completamente diferente. Este problema se puede atenuar mediante el uso de técnicas de combinación de métodos, como veremos en el apartado 2.

*** En inglés, *overfitting*.**

Figura 9. Ejemplos de regresión con árboles de decisión para los diferentes valores máximos de profundidad del árbol (max_depth)



1.4 Support Vector Regressor

El algoritmo de clasificación de las máquinas de vectores de apoyo (SVM) que vimos en el módulo 2 se puede modificar para resolver problemas de regresión. Este método se denomina Support Vector Regressor (SVR).

La regresión con vectores de apoyo (SVR) utiliza los mismos principios que las máquinas de vectores de apoyo (SVM) para la clasificación, solo que con algunos cambios menores. Nos conviene, pues, recordar brevemente la formulación *primal* del problema de optimización que introdujimos en el módulo 2 para aprender el hiperplano del margen máximo (w,b) de un clasificador SVM lineal:

$$minimizar: \ ||w||$$

$$sujeto\ a: \ y_i(\langle w,x_i\rangle + b) \geq 1, \forall\ 1 \leq i \leq N$$

donde w es un vector de pesos (con un componente para cada variable de entrada), b es un umbral (que está relacionado con la distancia del hiperplano en su origen), N es el número de ejemplos de entrenamiento y las y_i corresponden a las etiquetas de clase de cada ejemplo. Es decir, la función objetivo de un SVM es minimizar los pesos, más concretamente la norma L2 del vector de pesos w, con la restricción que cada punto del conjunto de datos de entrenamiento se tiene que situar en el lado correcto del hiperplano definido por (w,b).

En un Support Vector Regressor (SVR) lineal, la formulación *primal* toma la forma siguiente:

$$minimizar: \ ||w||$$

$$sujeto\ a: \ |y_i - (wx_i + b)| \leq \epsilon$$

Es decir, mantenemos el mismo objetivo de minimizar la norma L2 del vector de pesos w, pero cambiamos la restricción del problema para hacer que el error absoluto de las predicciones de la función de regresión ($\hat{y} = wx_i + b$) sea menor o igual que un margen predefinido, llamado *error máximo* (ϵ).

Con esta formulación que acabamos de hacer de un SVR, asumimos que hay una función lineal que aproxima todos los ejemplos del conjunto de datos de entrenamiento (x_i,y_i) con precisión ϵ. Pero, a veces, puede ser que no sea así y el problema de optimización convexa no es factible con esta formulación. De manera análoga a los SVM, introduciremos la idea del margen flexible para permitir ciertos errores en el modelo:

$$\begin{aligned} &minimizar: \ ||w|| + C\sum_{N}^{} \xi_i \\ &sujeto\ a: \ |y_i - (wx_i + b)| \leq 1\epsilon + \xi_i \end{aligned}$$

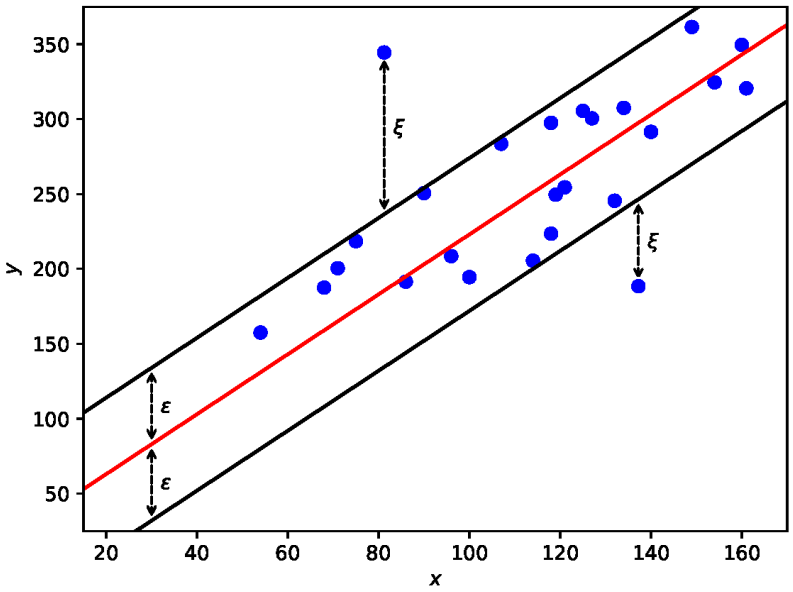
donde C es un hiperparámetro del modelo que controla la tolerancia a los errores.

Hiperparámetro C

Este hiperparámetro C no tiene ninguna relación con el número de clases C que hemos visto anteriormente en los problemas de clasificación.

La figura 10 muestra la salida de un modelo SVR. La línea roja representa la línea que se ajusta mejor a los datos y las líneas negras representan el margen de error (ϵ), que se ha fijado en un valor de 20 en este caso. Las variables ξ miden el coste de los errores en los ejemplos del conjunto de datos de entrenamiento. Fijaos que, por definición, el error es 0 para todos los puntos que residen dentro de la banda definida por ϵ.

figura 10. Ejemplo de función de regresión lineal unidimensional con banda de épsilon. Las variables ξ miden el coste de los errores en los puntos de entrenamiento que se encuentran fuera de la banda.



Tal y como vimos en el módulo 2, los clasificadores SVM dependen solo de un subconjunto de datos de entrenamiento (los vectores de apoyo), puesto que la función de coste no tiene en cuenta los puntos que se encuentran fuera del margen. Análogamente, el modelo de regresión SVR solo depende de un subconjunto de datos de entrenamiento, puesto que la función de coste ignora las muestras dentro de la banda definida por el hiperparámetro ϵ.

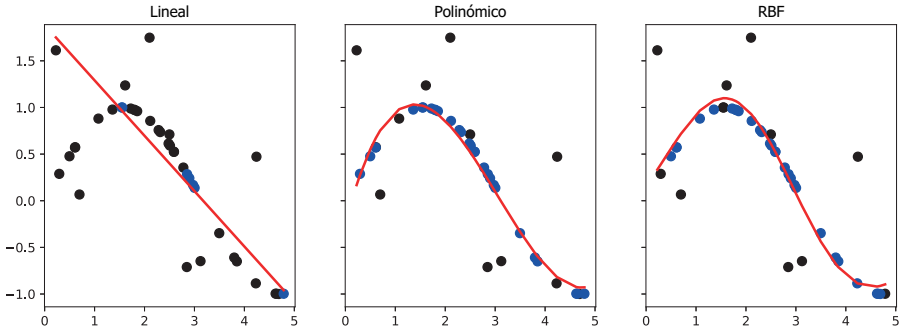
También vimos en el módulo 2 que, cuando en un problema de clasificación los datos no son separables linealmente, los SVM pueden utilizar la técnica del cambio de *kernel* para construir clasificadores no lineales. En la regresión con SVR también podremos hacer lo mismo. El algoritmo resultante es similar al SVR lineal, excepto en que cada producto escalar (wx_i) es reemplazado por una función *kernel* no lineal que proyecta los datos en un espacio de dimensionalidad más grande. Así, el algoritmo encuentra el hiperplano del margen máximo en el espacio de características transformado y la función de regresión resultante puede ser no lineal en el espacio de entrada original.

La figura 11 muestra tres ejemplos de regresión unidimensional en un conjunto de datos sintético utilizando, respectivamente, el SVR con núcleos lineales,

polinómicos y de función de base radial (RBF).* Los vectores de apoyo correspondientes se marcan en negro y el resto de datos, en azul.

* En inglés, *radial basis function*.

Figura 11. Ejemplos de regresión SVR, con núcleos lineales, polinómicos y de RBF, respectivamente



1.5 Redes neuronales

Cerraremos el apartado de modelos de regresión con una breve nota sobre el uso de las redes neuronales en las tareas de regresión. Como ya hemos mencionado en el módulo didáctico 2, las redes neuronales, especialmente en el contexto del aprendizaje profundo, son probablemente el algoritmo más utilizado hoy en día en el aprendizaje automático. Cuando tenemos conjuntos de datos complejos, con relaciones de entrada y de salida no lineales y disponemos de suficientes datos de entrenamiento, las redes neuronales son hoy en día un estándar *de facto* también en las tareas de regresión.

La teoría es exactamente la misma que se ha visto en el módulo 2. La única diferencia es que tenemos que usar una función de activación en la salida y una función de coste adecuadas para la tarea de regresión.

Ya hemos mencionado en el módulo 2 que, en caso de que la salida de una red neuronal sea un número real, la función de activación en la unidad de salida será normalmente lineal, es decir, $g(z) = z$. En algunos casos, cuando la salida deseada es un valor real acotado en un intervalo determinado, también se suele utilizar la función logística sigmoide, con la que se obtiene un rango de salida entre 0 y 1 que después se escala en el rango de salida esperado.

El error cuadrático medio (MSE) que hemos usado repetidamente a lo largo del apartado, también se suele utilizar como función de coste en redes neuronales para la regresión. Es una función derivable y nos permite (como toda función derivable) minimizar el error del modelo en el conjunto de los datos de entrenamiento. Por lo tanto, la función MSE es adecuada como función de coste para el método del descenso de gradientes. Hay que notar, sin embargo, que en el caso de las redes neuronales, la función MSE no será convexa y que, por lo tanto, el proceso de optimización puede ser más complejo.

2. Combinación de métodos

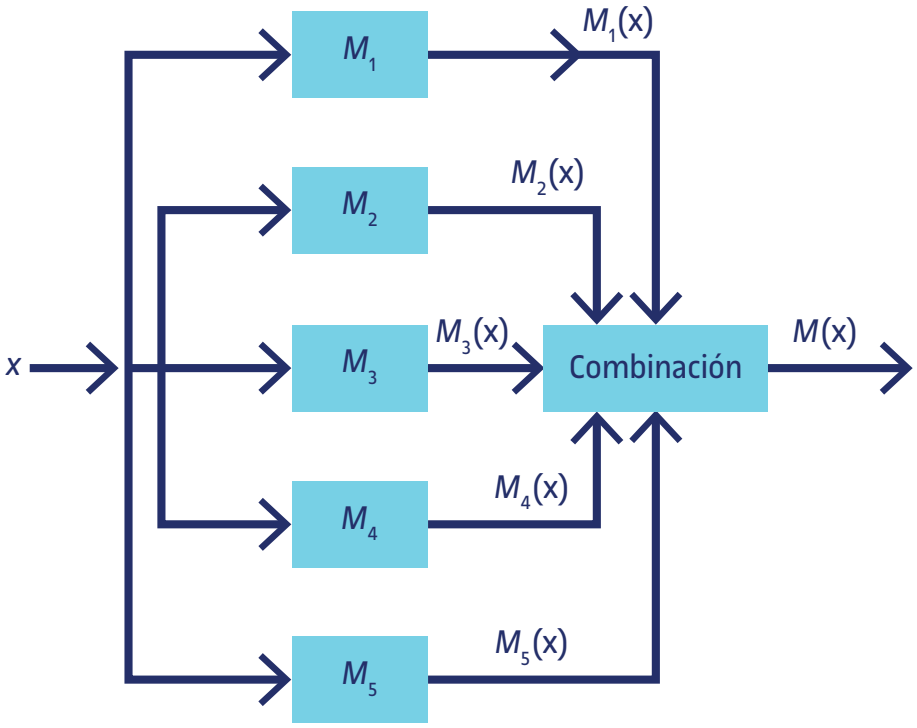
Ensemble learning

En inglés, la combinación de varios métodos de aprendizaje automático se conoce como *ensemble learning*.

En el módulo 2 y a lo largo de este módulo hemos considerado los diferentes métodos de aprendizaje supervisado de manera independiente. Una evolución que permite mejorar los resultados consiste en combinar diferentes métodos. Con esta visión, en lugar de trabajar con una única realización de un algoritmo, se entrenan múltiples algoritmos. Una vez hemos entrenado los algoritmos, podemos predecir el valor de salida para cada uno de ellos de manera independiente. Finalmente, podemos combinar las diferentes salidas para encontrar un resultado de consenso.

El esquema de un sistema formado por una combinación de métodos es el de la figura 12. Es decir, un conjunto de clasificadores (cinco en esta figura y denotados por $M_1,...,M_5$) en el que cada uno calcula su conclusión para una determinada entrada x. Una vez se conocen las conclusiones de los clasificadores, el sistema incluye un módulo para combinarlas a pesar de tomar una decisión final ($M(x)$ en la figura) que corresponde a la salida global del sistema para la entrada x.

Figura 12. Esquema de un conjunto de clasificadores



Es importante subrayar que, aunque se hable de un conjunto de clasificadores, este proceso también se aplica a problemas de regresión.

Los conjuntos de clasificadores se usan para reducir el error que hay asociado en los clasificadores cuando se utilizan de manera individual. Se ha comprobado que combinar más de uno ofrece mejores resultados, aunque los clasificadores se hayan construido con el mismo algoritmo.

2.1 Toma de la decisión final

Fijémonos en el problema de la clasificación para entender cómo funciona la combinación de métodos. La idea es que si tenemos, por ejemplo, cinco clasificadores, podemos tener uno o dos que fallan. Si la mayoría ofrece el resultado correcto, entonces el sistema en conjunto puede funcionar. Esto suponiendo que la manera de elegir la clase asociada a un objeto es seleccionar la más frecuente.

Sea cual sea la manera de decidir el valor final, todos los métodos que veremos se caracterizan por utilizar un conjunto de métodos. Cada uno de estos métodos produce una decisión débil a partir de la cual obtendremos la decisión final.

Imaginamos que queremos construir un sistema de combinación de métodos con un número N de métodos. Una vez tenemos los N modelos $M_1,M_2,...,M_N$ entrenados, cuando tenemos una nueva entrada x tenemos que determinar la salida. La salida del modelo M que combina todos los métodos será una combinación (mediante una función f) de los valores que devuelve cada uno de los modelos $M_1,M_2,...,M_N$ para el ejemplo x. Así, podemos expresar $M(x)$ por:

$$M(x) = f(M_1,M_2,...,M_N)$$

Decisión débil

En un sistema de clasificación binario, se define como *decisión débil* cualquier método que dé resultados por encima del 50 % de acierto. En el caso de que haya problemas de clasificación con C clases, entonces la probabilidad de acierto tendría que ser superior a $1/C$. En inglés, las decisiones débiles se conocen como *weak rules*.

Para calcular la combinación de los modelos, se pueden utilizar diferentes funciones de combinación f. Las más sencillas son las siguientes:

- En el caso de que haya problemas de regresión: como los valores $M_i(x)$ son numéricos, podemos definir f como la media de los valores $M_i(x)$. Es decir, $M(x) = \sum_i M_i(x)/N$

- En el caso de que haya problemas de clasificación: se selecciona la clase que aparece más a menudo entre los $M_i(x)$. Denominamos a esta metodología *hard voting* o *majority voting*.

A continuación, profundizamos en la decisión de la clase final con los problemas de clasificación. Hemos presentado el *hard voting* o *majority voting* como el método más sencillo. Ahora lo desarrollaremos y lo contrapondremos con el *soft voting*.

2.1.1 *Hard voting* o *majority voting*

Este es el método más sencillo de todos. Consiste en elegir la clase que ha recibido más votos como la final. Si tenemos N clasificadores y C clases, cada uno de estos clasificadores producirá una salida y_n, $n \in [1,N]$. Combinaremos las salidas de todos los clasificadores para contar las veces que cada clase ha sido escogida (o los votos que ha tenido cada clase), que definiremos como n_c, $c \in [1,C]$. Finalmente, la clase que haya sido elegida más veces (que tenga más votos) será la ganadora:

$$y = \text{argmax}(n_1,n_2,...,n_C)$$

Aquí, n_c representa cuantas veces una clase determinada ha sido escogida por el conjunto de los N clasificadores.

Pongamos un ejemplo. Imaginaos que queréis mejorar los resultados de una aplicación que clasifica automáticamente las flores teniendo en cuenta las medidas que envía el sistema láser: la longitud y la anchura del sépalo y del pétalo de cada flor. Recordad que el *dataset* consta de tres clases: «setosa», «versicolor» y «virginica». Si usamos el *majority voting*, crearemos N clasificadores a los que pasaremos los datos de entrada y nos darán N salidas. Pongamos, por ejemplo, que $N = 10$, es decir, que tenemos diez clasificadores y que sus salidas son las que se representan en la tabla 2.

Tabla 2. Resultados de los clasificadores con *majority voting*

Clasificador	Predicción
Clasificador 1	Versicolor
Clasificador 2	Versicolor
Clasificador 3	Setosa
Clasificador 4	Versicolor
Clasificador 5	Setosa
Clasificador 6	Virginica
Clasificador 7	Versicolor
Clasificador 8	Virginica
Clasificador 9	Setosa
Clasificador 10	Versicolor

Asignaremos los valores 1, 2 y 3 a las clases «setosa», «versicolor» y «virginica», respectivamente. Si contamos las veces que ha sido escogida cada clase, podemos comprobar que $n_1 = 3$, $n_2 = 5$ y $n_3 = 2$ (tabla 2). Por lo tanto, si aplicamos el argumento máximo tendremos que la clase ganadora es la 2, es decir, la «versicolor». Esta sería, pues, la salida final de nuestro modelo.

2.1.2 *Soft voting*

Ahora que hemos visto y entendido cómo funciona el *hard voting* podremos entender fácilmente el *soft voting*. Funciona exactamente igual que el anterior, pero, en lugar de tener clasificadores que producen una clase, tendremos un

vector de probabilidades. Esta es la única diferencia. Entonces, para combinar las diferentes predicciones y obtener la final, sumaremos todas las probabilidades correspondientes a cada clase y lo dividiremos por el número de clasificadores, como muestra la ecuación siguiente:

$$y = \text{argmax}\left(\frac{1}{N}\sum_{n=1}^{N}(p_{n,1},p_{n,2},...,p_{n,C})\right) = \text{argmax}\left(\frac{1}{N}(p_1,p_2,....,p_C)\right)$$

Aquí C es el número de clases, N es el número de clasificadores que hemos decidido utilizar y $p_{n,c}$ es la probabilidad del clasificador n para la clase c. Definimos p_c como la probabilidad acumulada de cada clase c que resulta de sumar las probabilidades $p_{n,c}$ de cada clasificador n por aquella clase c.

Continuamos con el ejemplo anterior, pero imaginamos que, en lugar de tener clasificadores que devuelven una clase, estos devuelven un vector de probabilidades que indica la posible pertenencia a cada clase. En este caso, la salida podría ser como la que se representa a la tabla 3, en la que cada columna indica la probabilidad de las clases «setosa», «versicolor» y «virginica».

Tabla 3. Resultados de los clasificadores con *soft voting*

Clasificador	Predicción Setosa	Predicción Versicolor	Predicción Virginica
Clasificador 1	0.10	0.50	0.40
Clasificador 2	0.00	0.65	0.35
Clasificador 3	0.50	0.30	0.20
Clasificador 4	0.25	0.45	0.30
Clasificador 5	0.55	0.35	0.10
Clasificador 6	0.20	0.35	0.45
Clasificador 7	0.05	0.65	0.30
Clasificador 8	0.05	0.40	0.55
Clasificador 9	0.45	0.35	0.20
Clasificador 10	0.25	0.50	0.25

En el *soft voting*, para obtener la predicción final, tendríamos que sumar las probabilidades de cada clase y dividirlas entre el número de clasificadores que tenemos. Siguiendo la tabla 3, podemos calcular que p_1 = 2.4, p_2 = 4.5 y, p_3 = 3.1 y sustituyendo a la fórmula anterior, obtenemos entonces que y = argmax((1/N)(2.4,4.5,3.1)) = 2. Por lo tanto, la predicción final será la clase 2, es decir, la clase «versicolor».

Tenemos, además, la posibilidad de asignar pesos a los diferentes clasificadores; esto se recomienda cuando hay clasificadores que consideramos mejores que otros.

2.2 Creación de los diferentes clasificadores

En el subapartado 2.1 hemos visto cómo se lleva a cabo la decisión conjunta a partir de un grupo de métodos. Aquí profundizaremos en varias estrategias para obtener el conjunto de métodos.

Para construir los diferentes clasificadores, hay varias estrategias, como las siguientes:

- **Combinación de diferentes algoritmos**: en este caso, cada método M_i de la figura 12 corresponde a un método diferente. En un ejemplo de regresión, para el que podríamos utilizar un método de regresión lineal, un kNN y un árbol de decisión. De este modo, podemos limitar los puntos débiles de un algoritmo con las fortalezas de otro. Estos tipos de combinaciones están hechos *ad hoc* para cada problema en concreto y requieren una exploración empírica para encontrar la mejor combinación de métodos.

- **Uso del mismo algoritmo**: en este caso, entrenamos a un único algoritmo, pero variando los datos de entrenamiento. Por ejemplo, en el caso de la regresión para determinar el precio de las casas, podríamos entrenar diferentes regresores lineales multivariantes utilizando un subconjunto de las diferentes variables (metros cuadrados, habitaciones, pisos, año de construcción). De este modo, en el caso de la figura 12, todos los métodos M_i serían regresores lineales, pero cada uno utilizaría datos diferentes para hacer la predicción del precio de la casa.

  En el supuesto de que optemos por la estrategia de utilizar el mismo algoritmo para los diversos clasificadores, podemos obtener los conjuntos de los diferentes datos de varias maneras:

  – Construcción de diferentes conjuntos de ejemplos: dado que los métodos construyen modelos basados en los ejemplos, los conjuntos de ejemplos diferentes generan clasificadores diferentes. Por lo tanto, para generar los clasificadores podemos generar varios conjuntos de entrenamiento a partir de un único conjunto de ejemplos. Estudiaremos los casos del *bagging* y del *boosting* como métodos que trabajan con la construcción de los diferentes conjuntos de ejemplos.

  – Manipulación de las características de entrada: corresponde a las manipulaciones sobre las variables o, en general, de las descripciones de los objetos. Un ejemplo de este caso lo encontramos cuando diferentes sistemas usan distintos conjuntos de variables. De todas maneras, hay que decir que este procedimiento no siempre funciona correctamente. Si las variables no están correlacionadas, los clasificadores pueden no tener toda la información necesaria para seleccionar correctamente la clase. Por lo tanto, el conjunto de clasificadores no funcionará correctamente. Veremos el *random subspace methods* y el *stacked learning* como ejemplos de manipulación de las características de entrada.

En este apartado nos centraremos en las metodologías que usan el mismo algoritmo, pero utilizando datos diferentes. En los próximos subapartados veremos en profundidad cuatro de las técnicas más utilizadas dentro del ámbito de la combinación de métodos: el *bagging*, el *boosting*, el *random subspace methods* y el *stacked learning*.

2.2.1 *Bagging*

Bagging

El nombre de *bagging* proviene de ***b**ootstrap aggregating*, que se refiere a la agregación de diferentes muestreos con repetición sobre los mismos datos.

Los algoritmos basados en el *bagging* dividen el conjunto de entrenamiento en T subconjuntos de muestras y aprenden un clasificador en cada uno de estos subconjuntos. Cada clasificador se construye de manera paralela e independiente del resto de clasificadores.

El *bootstrap* es un método estadístico para estimar una cantidad proveniente de varios subconjuntos de datos. Por ejemplo, si se quiere calcular la media de un conjunto muy grande de datos, primero se generan varios subconjuntos de estos, se calcula la media de cada subconjunto y, finalmente, todas las medias se agrupan también haciendo la media general. De este modo se consigue una estimación mejor que la media real.

Para explicarlo, partiremos de un conjunto de N muestras X, en el que $X = \{x_1,x_2,...,x_N\}$, con atributos o características $A = \{A_1,A_2,...,A_M\}$ y un *hashtag* asociado a cada muestra $L_i \in \{L_1,L_2\}$. El algoritmo base construye T subconjuntos de muestras de manera aleatoria y entrena a un clasificador débil M_t en cada subconjunto. La clasificación de una nueva muestra se hace aplicando cada uno de los T clasificadores al ejemplo y tomando la decisión final siguiendo una regla de agregación sobre los resultados de los T clasificadores. Normalmente, se usa una regla sencilla de *hard voting*, a pesar de que si tenemos clasificadores parciales que generan una determinada probabilidad, podemos establecer criterios probabilísticos más complejos. El algoritmo completo se muestra a continuación:

Bagging

Para cada iteración t_i ($t = 1,...,T$), hay que hacer lo siguiente:

1. Generar un subconjunto de muestras Z a partir de X.
2. Construir un clasificador M_t usando las muestras Z.
3. Añadir el clasificador a un conjunto de clasificadores que denominamos *ensemble*.

El algoritmo genera como salida cada uno de los clasificadores M_t.

La clasificación de nuevas instancias se hará aplicando cada uno de los T clasificadores. Por cada nueva muestra, escogeremos la clase que recibe más votos en el total de clasificadores.

La combinación de métodos mediante el *bagging* se aplica de forma análoga a los problemas de regresión. Los regresores se entrenan para cada subconjunto de muestras de forma independiente. La diferencia simplemente recae en la forma en que se decide el valor final, en la que se hace un promedio de los valores predichos por cada regresor en lugar de escoger la clase más predicha, en el caso de un conjunto de clasificadores, como hemos visto al subapartado 2.1.

Los algoritmos de *bagging*, a pesar de ser menos sofisticados que los de *boosting* (los veremos a continuación), son especialmente recomendables cuando el conjunto de entrenamiento contiene muestras mal etiquetadas.* Si en medio de la nube de puntos de una clase tenemos una muestra errónea, esta afectará solo a algunos de los clasificadores (aquellos en los que aparezca en el subconjunto correspondiente) y será neutra para la mayor parte de los clasificadores parciales.

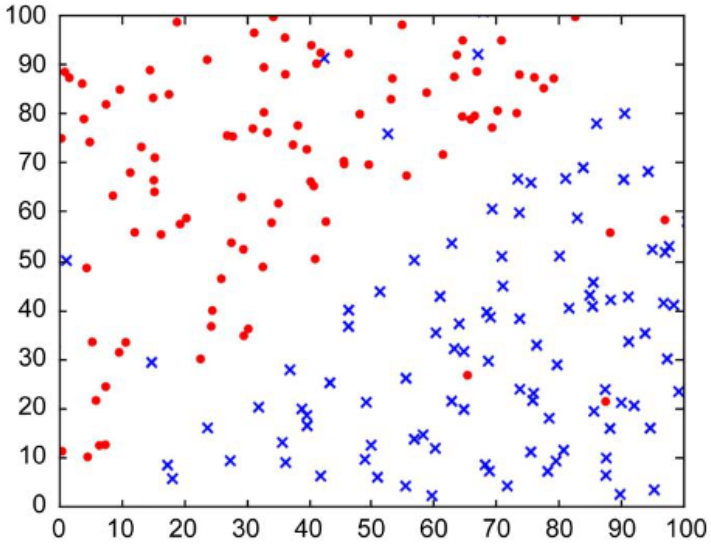
*** En inglés, *outliers*.**

Uno de los ejemplos más típicos de *bagging* es el algoritmo de *random forest*. En un *random forest* se aplica el *bootstrap* de forma que varios árboles de decisión se entrenan con diferentes subconjuntos de datos y cuando se hace una predicción, cada árbol la lleva a cabo individualmente y después se agrupan, de forma que se estima mejor el valor verdadero.

Ejemplo de aplicación

Para ilustrar el funcionamiento usaremos un problema sintético, en el que tendremos los conjuntos de entrenamiento y test representados en las figuras 13 y 14, respectivamente.

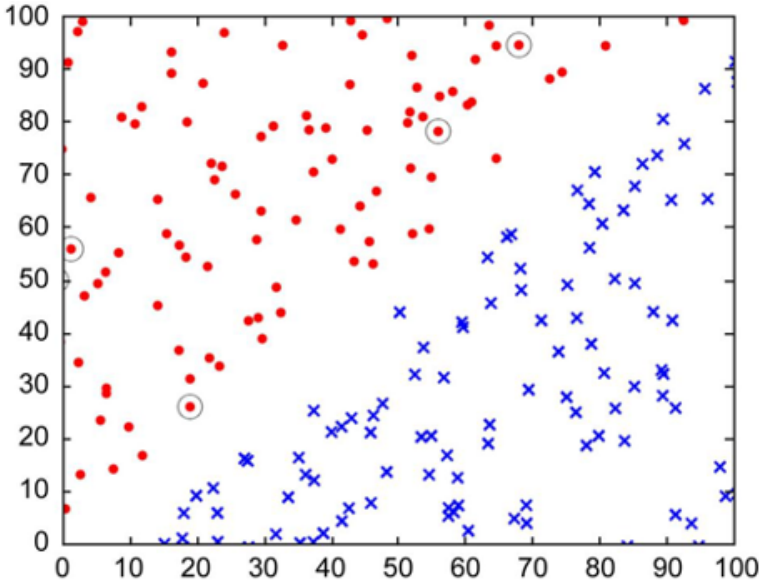
Figura 13. Muestras del conjunto de entrenamiento



Para hacer la clasificación, usaremos el algoritmo del vecino más próximo (kNN). Para cada muestra de test, buscaremos en el conjunto de entrenamiento cual es la muestra más próxima y asignaremos a la muestra de test el *hashtag* de este vecino más próximo (utilizaremos, por lo tanto, $k = 1$ o 1NN).

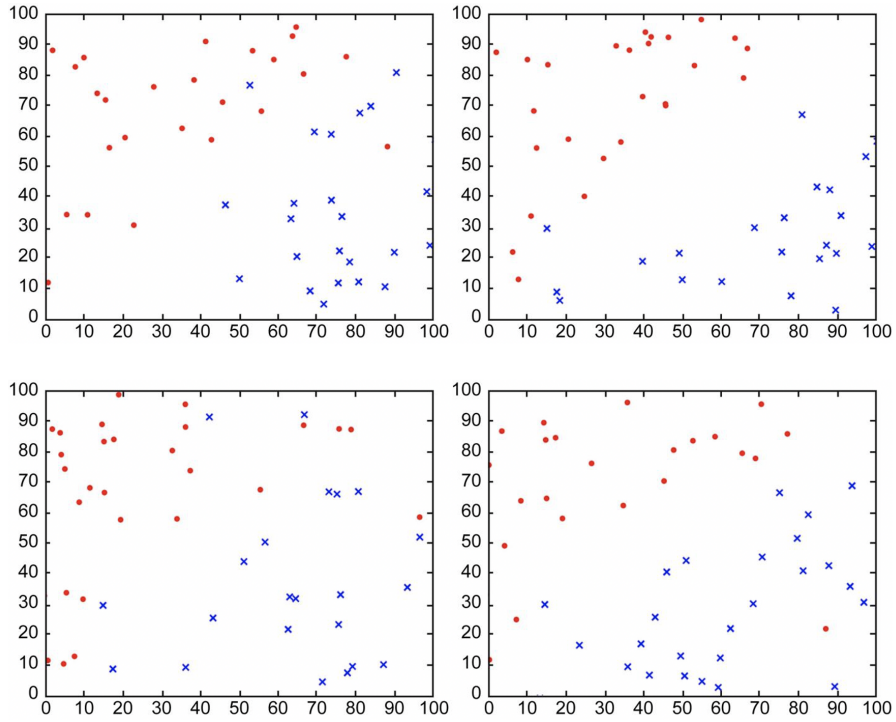
Podéis observar que el conjunto de entrenamiento es linealmente separable, si no fuera por la presencia de algunos ejemplos probablemente mal etiquetados. Este tipo de muestras a menudo hacen bajar considerablemente el rendimiento del vecino más próximo. Hemos trazado un círculo negro en los ejemplos que estarían mal clasificados en este conjunto de test.

Figura 14. Muestras del conjunto de test. Los puntos de test están representados con el color (y símbolo) de la clase de *ground truth*. Hemos marcado algunos puntos rojos con un círculo alrededor para indicar que se trata de ejemplos que se clasificarían erróneamente como cruces azules si utilizáramos el algoritmo 1NN con los datos de entrenamiento de la figura 13.



El objetivo es, pues, aplicar las técnicas de *bagging* que permitan hacer una combinación de clasificadores que aísle estos *outliers* y elimine la influencia. Aplicaremos un total de cien iteraciones, que harán cien muestreos aleatorios del conjunto de entrenamiento. En la figura 15 mostramos algunos de estos conjuntos.

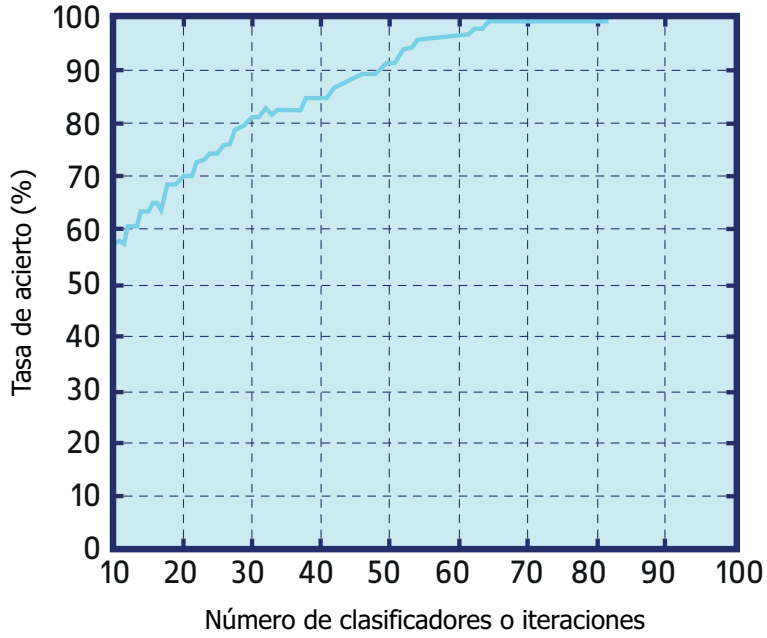
Figura 15. Muestras de entrenamiento seleccionadas aleatoriamente en cuatro iteraciones del algoritmo



Podéis observar que la presencia de los *outliers* se reparte en las diferentes iteraciones y es muy difícil que una muestra mal etiquetada haga que «se equivoquen» la mayoría de los clasificadores.

Para completar el análisis, hemos calculado la precisión acumulada en las muestras del conjunto de test teniendo en cuenta el número de iteraciones (figura 16). Como era de esperar, el error decrece a medida que incrementamos el número de clasificadores parciales, hasta llegar a eliminar completamente el efecto de los *outliers*.

Figura 16. Porcentaje de acierto en función del número de clasificadores parciales



2.2.2 *Boosting*

Los algoritmos de *boosting* se basan en combinar un conjunto de clasificadores que se han entrenado teniendo en cuenta los diferentes subconjuntos de muestras de entrenamiento. A diferencia del *bagging*, en el que los clasificadores se construyen de manera paralela, en el *boosting* los clasificadores se construyen de manera secuencial, asociando pesos a las muestras relacionados con su dificultad. Así pues, cada iteración se centra en los elementos más difíciles de clasificar de las iteraciones anteriores.

Boosting

El nombre de *boosting* se basa en la pregunta planteada por Kearns y Valiant (1988 y 1989): «¿Un grupo de clasificadores débiles puede crear un clasificador sólido?» Los algoritmos que pueden conseguir esta condición ampliando (*boosting*) rápidamente los resultados de los clasificadores débiles fueron denominados *boosting*.

Hay varias maneras de elegir estas muestras según el algoritmo concreto utilizado. En este subapartado veremos el algoritmo llamado AdaBoost.

El nombre AdaBoost proviene de la contracción de las palabras *adaptive* y *boosting* y describe un procedimiento que permite construir, de manera incremental, un sistema de clasificadores y un vector de pesos que sirven para combinar estos clasificadores.

Este método está específicamente diseñado para combinar reglas. Del mismo modo que en el algoritmo del *bagging*, se basa en la combinación lineal de muchas reglas débiles para crear un clasificador muy robusto con un error arbitrariamente bajo en el conjunto de entrenamiento. A diferencia del *bagging*, estas reglas débiles se aprenden secuencialmente manteniendo una distribución de pesos sobre los ejemplos de entrenamiento. Estos pesos se irán actualizando a medida que vayamos adquiriendo nuevas reglas. Esta actualización depende de la dificultad de aprendizaje de los ejemplos de entrenamiento. Es decir, los pesos de los ejemplos serán directamente proporcionales a su dificultad de aprendizaje y, por tanto, la adquisición de nuevas reglas se irá dirigiendo a los ejemplos con más peso (y dificultad).

Lectura complementaria

El algoritmo AdaBoost fue propuesto en **Y. Freund; R. E. Schapire** (1997). «A Decision –Theoric Generalization of Online Learning and an Application to Boosting». *Journal of Computer and System Sciences* (vol. 1, n.º 55).

Referencia bibliográfica

Y. Freund; R. E. Schapire (1999). «A Short Introduction to Boosting». *Journal of Japanese Society for Artificial Intelligence* (vol. 5, n.º 14, págs. 771-780).

El AdaBoost es un algoritmo que pretende obtener una regla de clasificación muy precisa combinando muchos clasificadores débiles, cada uno de los cuales obtiene una precisión moderada. Este algoritmo trabaja eficientemente con espacios de atributos muy grandes y ha sido aplicado con éxito en muchos problemas prácticos. A continuación se muestra el pseudocódigo del algoritmo:

algoritmo AdaBoost

entrada: $S = \{(x_i, y_i), \forall i = 1..N\}$

S es el conjunto de ejemplos de entrenamiento
D_1 es la distribución inicial de pesos
N es el número de ejemplos de entrenamiento
$y_i \in \{-1, +1\}$

$D_1(i) = \frac{1}{N}, \forall i = 1..N$

Se hacen T iteraciones:

para $t := 1$ **hasta** T **hacer**

Se obtiene la hipótesis débil $h_t : X \to \{-1, +1\}$

$h_t = \texttt{ObtenerHipótesisDébil}(X, D_t)$

$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$

$\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$

Se actualiza la distribución D_t

$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}, \forall i = 1..N$

Z_t es un factor de normalización tal que D_{t+1} sea una
distribución, por ejemplo: $Z_t = \sum_{i=1}^{N} D_t(i)\exp(-\alpha_t y_i h_t(x_i))$

fin para

devuelve la combinación de hipótesis: $f(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$

fin AdaBoost

Las hipótesis débiles se aprenden secuencialmente. En cada iteración se aprende una regla que se sesga para clasificar los ejemplos con más dificultades por parte del conjunto de reglas precedentes. AdaBoost mantiene un vector de pesos como una distribución D_t sobre los ejemplos. En la iteración t, el objetivo del algoritmo es encontrar una hipótesis débil, $h_t : X \to \{-1, +1\}$, con un error moderadamente bajo en cuanto a la distribución de pesos D_t. En este marco, las hipótesis débiles $h_t(x)$ dan como predicciones valores reales que correspon-

den a valores de confianza. Inicialmente, la distribución de pesos D_1 es uniforme y, en cada iteración, el algoritmo de *boosting* incrementa (o disminuye) exponencialmente los pesos $D_t(i)$ en función de si $h_t(x_i)$ hace una predicción buena (o mala). Este cambio exponencial depende de la confianza $|h_t(x_i)|$. La combinación final de la hipótesis, $h_t : X \to \{-1, +1\}$, calcula sus predicciones ponderando con pesos los votos de las diferentes hipótesis débiles:

$$f(x) = \sum_{t=1}^{T} \alpha_t \cdot h_t(x)$$

Para cada ejemplo x, el signo de $f(x)$ se interpreta como la clase predicha (el AdaBoost básico trabaja solo con dos clases de salida, –1 o +1) y la magnitud $|f(x)|$ como una medida de la confianza de la predicción. Esta función se puede usar para clasificar nuevos ejemplos o para hacer un ranquin en función de un grado de confianza.

Ejemplos de aplicación

Supongamos que queremos hacer una aplicación para teléfonos móviles que ayude a los aficionados de la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero.

El algoritmo AdaBoost tiene dos parámetros para determinar el número máximo de iteraciones y la manera de construcción de las reglas débiles. En este subapartado veremos la aplicación del algoritmo escogiendo como conjunto de entrenamiento los datos de la tabla 4 y como test, el de la tabla 5. Fijamos el número máximo de iteraciones en 5 y como reglas débiles, las reglas del estilo: *atributo* = *valor* y *atributo* ≠ *valor*.

Tabla 4. Conjunto de entrenamiento

class	ninguno-shape	cap-color
+1	convexo	brown
–1	convexo	yellow
–1	bello	white
+1	convexo	white

Font: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010).

Tabla 5. Ejemplo de test

class	ninguno-shape	cap-color
–1	bello	yellow

Font: problema «mushroom» del repositorio UCI (Frank y Asunción, 2010).

Como primer paso del algoritmo, inicializamos los cuatro elementos del vector de pesos D_1 a 0.25. A continuación, empezamos el proceso iterativo. El primer paso es calcular la regla débil. La tabla 6 muestra las diferentes reglas aplicables* y su error. La regla *cs* = *bell* asignará las predicciones –1, –1, +1 y –1, respectivamente (se asigna la predicción +1 cuando la regla se cumple y, –1, en caso contrario). Para calcular el error, sumamos el peso $D_t(i)$ de los ejemplos a

*** En el que *cs* corresponde a *cap-shape* y *cc* a *cap-color*.**

las predicciones de los que sean diferentes a sus clases. En este caso sumaríamos el peso del primer, tercer y cuarto ejemplo (en total, 0.75). El resto de los casos los calcularíamos de manera similar.

Tabla 6. Reglas débiles

Regla	Error
cs = bell	0.75
cs ≠ bell	0.25
cs = convex	0.25
cs ≠ convex	0.75
cc = brown	0.25
cc ≠ brown	0.75
cc = white	0.5
cc ≠ white	0.5
cc = yellow	0.75
cc ≠ yellow	0.25

Escogemos como regla débil una de las que tengan un error mínimo (en el ejemplo hemos escogido *cs ≠ bell*). Así, el error de la primera iteración es $\epsilon_1 = 0.25$ y, por tanto, $\alpha_1 = \frac{1}{2}\ln(\frac{1-\epsilon_1}{\epsilon_1}) = 0.5493$.

El paso siguiente consiste en actualizar el vector de pesos. Empezamos por calcular $D_1(i)\exp(-\alpha_1 y_i h_1(x_i))$. La tabla 7 muestra el valor de esta expresión para los cuatro ejemplos y el valor de todos los componentes. La última columna muestra los valores finales del vector de pesos para la segunda iteración. Podemos ver como el segundo elemento, que es donde el modelo comete el error, es el que pasa a tener el valor D_2 más grande, mientras que el valor se reduce en el resto de los elementos predichos correctamente.

Tabla 7. Cálculo de D_2

.

i	$D_1(i)$	α_1	y_i	$h_1(x_i)$	*numerador*	$D_2(i)$
1	0.25	0.5493	+1	+1	0.1443376	0.1667
2	0.25	0.5493	–1	+1	0.4330127	0.5
3	0.25	0.5493	–1	–1	0.1443376	0.1667
4	0.25	0.5493	+1	+1	0.1443376	0.1667

El resto de iteraciones se calcularían de manera similar. La tabla 8 muestra los resultados de las cinco iteraciones del entrenamiento.

Tabla 8. Evolución de los pesos

t	α_t	$regla_t$	D_t
1	0.549	*cs ≠ bell*	(0.25,0.25,0.25,0.25)
2	0.805	*cc = brown*	(0.167,0.5,0.167,0.167)
3	1.099	*cc ≠ yellow*	(0.999,0.3,0.099,0.499)
4	0.805	*cs ≠ bell*	(0.056,0.167,0.5,0.278)
5	0.805	*cc = brown*	(0.033,0.5,0.3,0.167)
6			(0.019,0.3,0.18,0.5)

El clasificador da una predicción correcta sobre el ejemplo que muestra la tabla 5. Para obtener la predicción, tenemos que calcular:

$$\text{signo}(\textstyle\sum_{t=1}^{5} \alpha_t h_t(x)) =$$

$$= \text{signo}(0.549 \times (-1) + 0.805 \times (-1) +$$

$$+1.099 \times (-1) + 0.805 \times (-1) + 0.805 \times (-1)) =$$

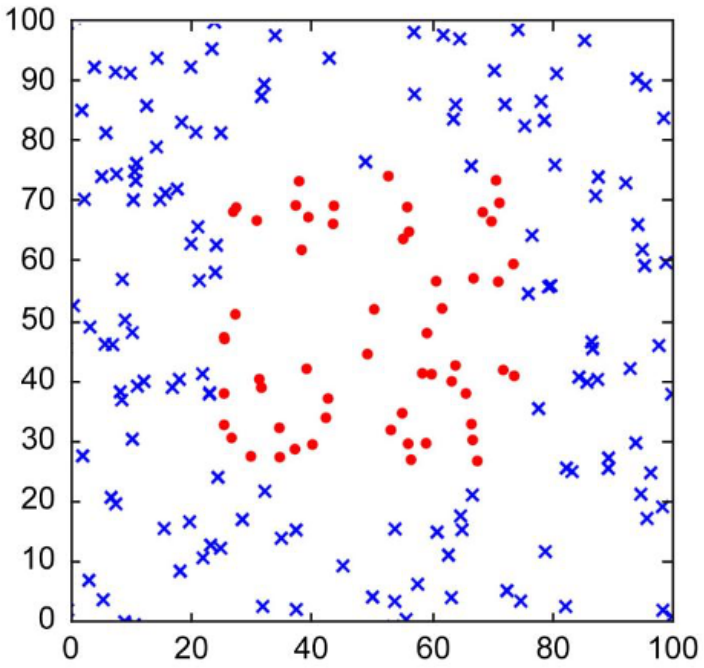
$$= \text{signo}(-4.063) = -1$$

En el ejemplo que hemos visto, tenemos características que seleccionamos según si tienen un valor concreto dentro de un conjunto discreto de posibles valores, con reglas del estilo *atributo* = *valor* y *atributo* $\neq$ *valor*. Ahora veremos otro ejemplo en el que los atributos toman valores reales continuos y, por lo tanto, definiremos las reglas del estilo *atributo* $\geq$ *valor* o *atributo* $\leq$ *valor*; esto se denomina *clasificador débil basado en un umbral*.

El clasificador débil basado en un umbral elige una característica de los datos de entrenamiento y selecciona el umbral que minimiza el error en aquella característica. Se asigna la clase 1 a la muestra si la característica es más grande que el umbral (y 0 cuando no lo es). Podéis observar que este clasificador nos dará siempre un porcentaje de clasificación superior al 50 %; en caso contrario, solo tenemos que dar la vuelta al umbral.

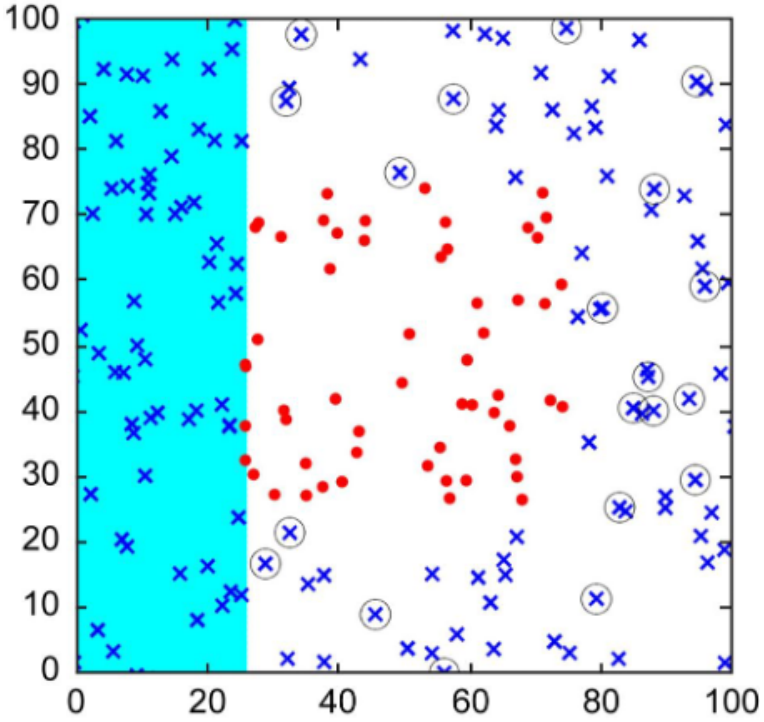
Partimos del conjunto de datos que se muestran en la figura 17.

Figura 17. Ejemplos de entrenamientos pertenecientes a dos clases



A continuación, en la figura 18 daremos un paso en el algoritmo y marcaremos en color azul claro la región que el AdaBoost nos daría actualmente como clase cruz (la región blanca pasa a ser clase punto).

figura 18. Resultado de la primera iteración del algoritmo AdaBoost.



Como se puede observar, el algoritmo ha puesto un umbral a la primera coordenada y pasa a ser clase punto todo lo que está a la derecha del valor 25 (eje X). Con este primer paso, conseguimos clasificar correctamente todos los puntos rojos y la parte inicial de las cruces azules.

Para ilustrar el funcionamiento del método, hemos añadido un círculo negro a los puntos que tienen un peso alto en esta iteración (con un historial largo de clasificación incorrecta).

En la siguiente iteración (figura 19), el algoritmo ha generado un nuevo clasificador débil que focaliza los puntos de la derecha que tenían un peso muy alto. La combinación de las dos reglas de clasificación sencillas nos da una frontera no lineal que sigue acertando en todos los puntos rojos y falla en los puntos centrales. Podéis observar que es en este espacio donde localizamos ahora las regiones de máximo peso en las que podemos focalizar el clasificador siguiente.

A continuación, en la figura 20, se puede observar como el clasificador débil ha elegido la segunda característica (eje Y) para poner el umbral. Con la unión de los tres clasificadores débiles, la región de pertenencia se vuelve más no lineal y engloba ya los puntos de la parte inferior.

Figura 19. Resultado de la segunda iteración del algoritmo AdaBoost

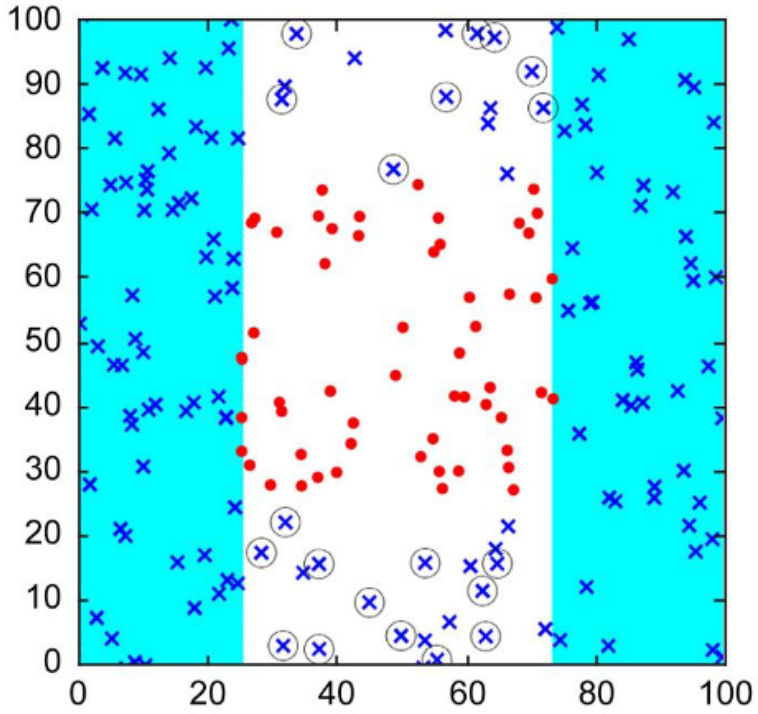
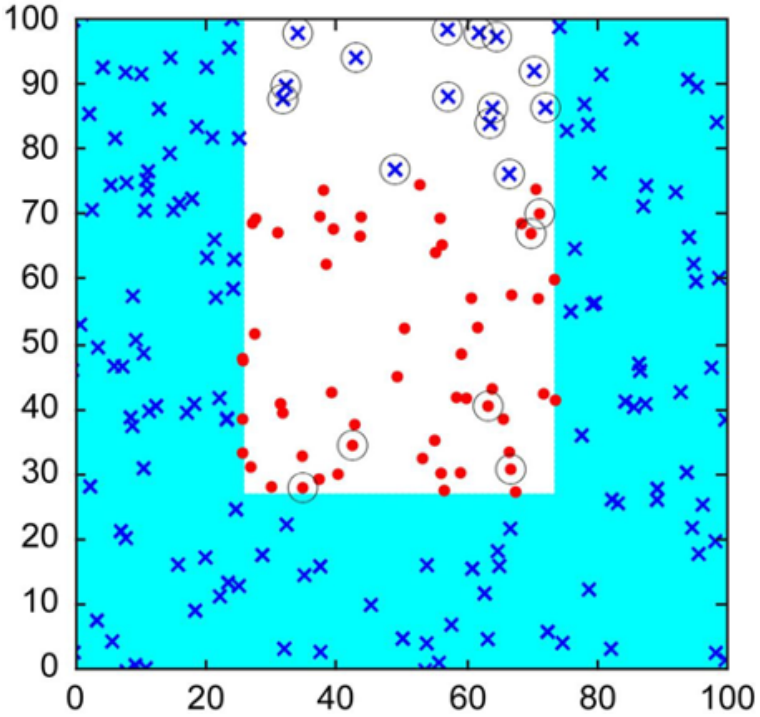
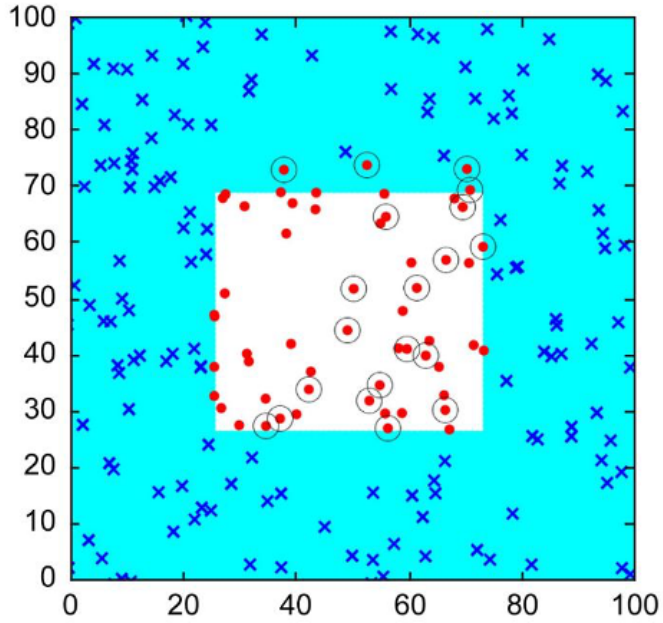


Figura 20. Resultado de la tercera iteración del algoritmo AdaBoost



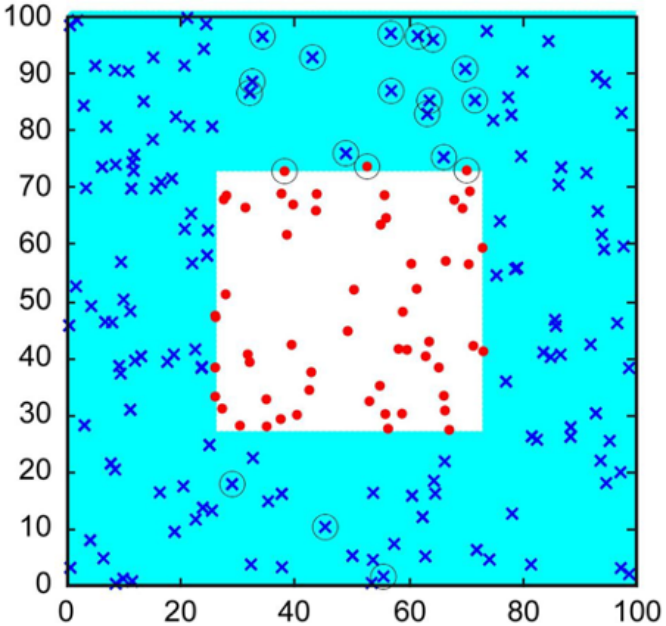
En la iteración siguiente (figura 21), el clasificador débil se ha focalizado en los puntos de la región superior. Podéis notar que el error de entrenamiento todavía no es cero, puesto que el último clasificador ha empeorado los puntos que anteriormente estaban muy bien clasificados. Este es un caso frecuente en el entrenamiento del Adaboost que requiere más iteraciones para encontrar una frontera de separación con un error de entrenamiento mínimo.

Figura 21. Resultado de la cuarta iteración del algoritmo AdaBoost



Si continuamos iterando el algoritmo, llegaremos a una solución final como la que se muestra a la figura 22.

Figura 22. Resultado de hacer veinte iteraciones del algoritmo AdaBoost



La combinación de los clasificadores simples (biparticiones del espacio) permite generar fronteras de decisión altamente no lineales, de manera iterativa y con un coste computacional bajo, por lo que acontece en uno de los clasificadores más utilizados actualmente.

Análisis del método

Las ventajas que nos aporta este método son diversas. En primer lugar, tiene pocos parámetros: el número máximo de iteraciones y la forma de las hipótesis débiles. La implementación de este algoritmo es relativamente sencilla.

Por otro lado, al finalizar el proceso, nos da información muy útil. De las reglas generadas se puede extraer conocimiento que puede ser interpretado por humanos. Los pesos finales indican qué ejemplos no se han podido clasificar correctamente. Esto nos da información de posibles *outliers* o de posibles errores de etiquetado.

El principal inconveniente que tiene es su coste computacional. Se han dedicado esfuerzos para tratar este tema, como la versión LazyBoosting,* en la que en cada iteración solo se explora un subconjunto aleatorio de las posibles reglas débiles. En este trabajo se reporta una reducción sustancial del coste computacional con una reducción mínima de los resultados.

* **G. Escudero; L. Màrquez; G. Rigau (2000). «Boosting Applied to Word Sense Disambiguation». En:** ***Proceedings of the 12th European Conference donde Machine Learning.***

El AdaBoost es un algoritmo teóricamente muy fundamentado. El error de generalización de la hipótesis final puede ser acotado en términos del error de entrenamiento. Este algoritmo está basado en la maximización del margen, lo que hace que tenga muchas similitudes con las máquinas de vectores de apoyo.

Lectura complementaria

El Adaboost.mh fue propuesto en: **R. E. Schapire; Y. Singer** (2000). «Boostexter: A Boosting-based System for Text Categorization». *Machine Learning* (vol. 39, n.º 2-3).

El algoritmo básico solo permite clasificar en dos clases. Existen variantes para tratar el problema multiclase, como es el caso del Adaboost.mh.

2.2.3 Algoritmos *random subspace methods*

En el caso de los *random subspace methods* (RSM), partimos de un enfoque similar al *bagging*, pero en este caso los subconjuntos de datos no se hacen sobre el espacio de las muestras, sino en el espacio de sus atributos. Partiendo de un conjunto de N muestras X, donde X = $\{x_1,x_2,...,x_N\}$, con atributos o características A = $\{A_1,A_2,...,A_M\}$ y un *hashtag* asociado a cada muestra $L_i \in \{L_1,L_2\}$, generaremos T subconjuntos diferentes. A cada uno de ellos les pondremos todos los ejemplos, con la particularidad de que eliminaremos aleatoriamente algunos de sus atributos. La selección de los atributos será diferente en cada iteración y se obtendrán T clasificadores parciales que se han entrenado en diferentes subespacios de las muestras originales. El algoritmo completo se detalla a continuación:

RSM

Para cada iteración t_i ($t = 1,...,T$) hay que hacer lo siguiente:

1. Coger un subconjunto de atributos K correspondiente a las muestras X.
2. Construir un clasificador M_t usando las muestras en el espacio reducido.
3. Añadir el clasificador al conjunto de clasificadores que denominamos *ensemble*.

El algoritmo genera como salida cada uno de los clasificadores M_t.

Los RSM son útiles en los problemas en los que hay pocas muestras y estas se encuentran empotradas en un espacio de alta dimensionalidad o bien, en problemas de alta dimensionalidad en los que hay mucha redundancia en los atributos o características. Un ejemplo sencillo de RSM en el problema de recolección de setas seria utilizar un primer clasificador que solo usara el atributo *cap-shape* y un segundo clasificador que utilizara solo el *cap-color*.

2.2.4 *Stacked generalization*

En el caso del *stacked generalization*, partimos de un enfoque similar al del *bagging*, pero en este caso los subconjuntos de datos no se hacen sobre el espacio de las muestras, sino en el espacio de sus atributos, tal y como sucede con los RSM vistos anteriormente.

Concretamente, los algoritmos de *stacked generalization* o *stacked learning* amplían el conjunto de características o atributos a la hora de entrenar el conjunto de clasificadores que formarán la combinación de métodos (*ensemble*), teniendo en cuenta las decisiones tomadas por los clasificadores anteriores.

Cada clasificador débil genera una decisión sobre el *hashtag* de cada muestra y este *hashtag* está concatenado o apilado* a continuación de los datos y es un atributo más. Los detalles del algoritmo los podemos encontrar a continuación:

* En inglés, *stacked*.

Stacked learning
Para cada iteración t_i ($t = 1,...,T$) hay que hacer lo siguiente:
1. Generar una predicción sobre el *hashtag* de cada muestra x_i, obteniendo una pseudoetiqueta l_i.
2. Construir el conjunto extendido de muestras X, en el que cada elemento x_i de este conjunto consiste en la muestra x_i anterior concatenada con la pseudoetiqueta l_i generada: $x_i = [x_{i,1},x_{i,2},...,x_{i,M},l_i]$.
El algoritmo genera como salida cada uno de los clasificadores M_t de cada iteración.
La clasificación de nuevas instancias se hará aplicando cada uno de los t clasificadores teniendo en cuenta que hay que añadir a cada iteración t la característica obtenida en la iteración $t - 1$.

A continuación, veremos un ejemplo para aclarar el funcionamiento del *stacked learning*. Imagináos, volviendo al ejemplo de las flores, que queréis mejorar el comportamiento de vuestro modelo. Habéis estado haciendo pruebas con el algoritmo de máquinas de vectores de apoyo con diferentes *kernels* e hiperparámetros. Os habéis dado cuenta de que mientras que uno de los modelos hace un buen trabajo clasificando la clase «versicolor», pero no lo hace nada bien ni con «setosa» ni con «virginica». A su vez, otro de los modelos entrenados

consigue una buena precisión con la clase «setosa», pero no con las otras. Y finalmente, un tercer modelo consigue una buena precisión únicamente con la clase «virginica».

Este ejemplo no es nada extraño; al contrario, se suele dar con frecuencia en el ámbito del aprendizaje automático, puesto que cada algoritmo tiene unas ventajas y unos defectos. Entonces, lo que quiere el *stacked learning* es precisamente combinar estas predicciones hechas por clasificadores (conocidos como *débiles*) y usarlas como entrada de un algoritmo (conocido como *metalgoritmo*), que las utiliza para hacer una mejor predicción.

El modelo de aprendizaje mostrado es lo más simple posible. Los sistemas de *stacked learning* han tenido múltiples aplicaciones en los problemas de aprendizaje sobre los datos secuenciales y los modelos gráficos.

Bibliografía

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. EE. UU.: Springer.

Duda, R. O.; Hart, P. E.; Stork, D. G. (2001). *Pattern Classification* (2.ª ed.). EE. UU.: John Wiley and Sonidos, Inc.

Frank, A.; Asuncion, A. (2010). *UCI Machine Learning Repository* (Disponible en línea). EE. UU.: University of California, School of Information and Computer Science. [Fecha de consulta: 22 de enero de 2018.]
<http://archive.ics.uci.edu/ml>

Géron, A. (2017). *Hands-Donde Machine Learning with Scikit-Learn and TensorFlow*. EE. UU.: O'Reilly Media.

Goodfellow, I.; Bengio, Y.; Courville, A. (2016). *Deep Learning*. EE. UU.: MIT Press.

Gulli, A.; Palo, S. (2017). *Deep Learning with Keras*. EE. UU.: Packt Publishing.

Mitchell, T. M. (1997). *Machine Learning*. EE. UU.: McGraw-Hill.

Segaran, T. (2007). *Programming Collective Intelligence*. EE. UU.: O'Reilly.

Shawe-Taylor, J.; Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. RU: Cambridge University Press.