Métodos supervisados

Jordi Casas Roma
Julià Minguillón Alfonso

1 crédito
xxx/xxxxx/xxx

Índice

Introducción

De forma complementaria, en el quinto bloque se describen los principales métodos de aprendizaje supervisado, formado por algoritmos que requieren un conjunto de entrenamiento etiquetado con las clases o variables objetivo del análisis. En esta parte se verá el algoritmo k-NN, las máquinas de soporte vectorial, las redes neuronales, los árboles de decisión y los métodos probabilísticos.

Objetivos

En los materiales didácticos de este módulo encontraremos las herramientas indispensables para asimilar los siguientes objetivos:

1. XXX

1. Introducción

De forma complementaria, en el quinto bloque se describen los principales métodos de aprendizaje supervisado, formado por algoritmos que requieren un conjunto de entrenamiento etiquetado con las clases o variables objetivo del análisis. En esta parte se verá el algoritmo k-NN, las máquinas de soporte vectorial, las redes neuronales, los árboles de decisión y los métodos probabilísticos

Recordar implicacions supervisat

Recordar notació de dades D

2. Algoritmo k-NN

El k-NN o k vecinos más cercanos (en inglés, k *nearest neighbours*) es un algoritmo de aprendizaje supervisado de clasificación, de modo que a partir de un juego de datos de entrenamiento, su objetivo será clasificar correctamente todas las instancias nuevas. El juego de datos típico de este tipo de algoritmos está formado por varios atributos descriptivos y un solo atributo objetivo, también llamado *clase*.

2.1. Funcionamiento del método

En contraste con otros algoritmos de aprendizaje supervisado, k-NN no genera un modelo fruto del aprendizaje con datos de entrenamiento, sino que el aprendizaje sucede en el mismo momento en el que se pide clasificar una nueva instancia. A este tipo de algoritmos se les llama algoritmos de aprendizaje perezoso o *lazy learning methods*, en inglés.

El funcionamiento del algoritmo es muy simple. Para cada nueva instancia a clasificar, se calcula la distancia con todas las instancias de entrenamiento y se seleccionan las k instancias más cercanas. La clase de la nueva instancia se determina como la clase mayoritaria de sus k instancias más cercanas.

El funcionamiento del método se detalla en el Algoritmo 1 y se describe a continuación:

1) Fijamos una valor para k, habitualmente pequeño.

2) Dada una nueva instancia x del juego de datos de prueba, el algoritmo selecciona las k instancias del juego de datos de entrenamiento más cercanas, de acuerdo con la métrica de similitud utilizada.

3) Finalmente, se asigna la instancia x a la clase más frecuente de entre las k instancias seleccionadas como más cercanas.

Sin duda se trata de un algoritmo tremendamente simple y a la par, con un nivel de efectividad similar a otros algoritmos más complejos y elaborados.

Veamos gráficamente, mediante la Figura 1, las distintas clasificaciones que se obtienen al variar el valor de k. Supongamos que debemos clasificar una nueva instancia, representada aquí como una bola blanca, utilizando un conjunto de datos de entrenamiento binario, representado por bolas de color gris o negro.

Algoritmo 1 Pseudocódigo del algoritmo *k-NN*

Require: k (número de vecinos) y x (nueva instancia a clasificar)

for all $d_i \in D$ **do**

Calcular la distancia $d(d_i, x)$

end for

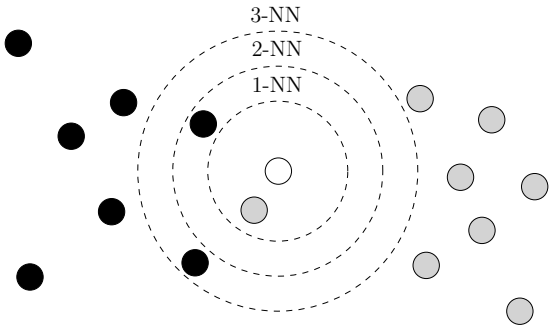
Inicializar I con las clases c_i de las k instancias de entrenamiento más próximas a x.

return La clase mayoritaria en I

Según el valor de k, la nueva instancia será clasificada como:

- Para $k = 1$ el algoritmo clasificará la nueva instancia como clase "gris".
- Para $k = 2$ el algoritmo no tiene criterio para clasificar la nueva instancia.
- Para $k = 3$ el algoritmo clasificará la nueva instancia como clase "negra".

Figura 1. Ejemplo de valores de k en el algoritmo k-NN



Su mayor debilidad es la lentitud en el proceso de clasificación, puesto que su objetivo no es obtener un modelo optimizado, sino que cada instancia de prueba es comparada contra todo el juego de datos de entrenamiento. Será la bondad de los resultados lo que determinará el ajuste de aspectos del algoritmo como el propio valor k, el criterio de selección de instancias para formar parte del juego de datos D de entrenamiento o la propia métrica de medida de similitud.

2.2. Detalles del método

Este método presenta dos variables importantes que pueden determinar la bondad del método. Éstas son, la métrica de similitud escogida y al valor de la propia k.

La métrica de similitud utilizada debería tener en cuenta la importancia relativa de cada atributo, puesto que ésta influirá fuertemente en la relaciones de cercanía que se irán estableciendo en el proceso de construcción del algoritmo. La métrica de distancia puede llegar a contener pesos que nos ayudarán a calibrar el algoritmo de clasificación, convirtiéndola de hecho en una métrica personalizada. Además, debería ser eficiente computacionalmente, ya que deberemos de ejecutar el cálculo de similitud multitud

de veces durante el proceso de clasificación de nuevas instancias.

Una de las distancias más utilizadas es la euclídea, especialmente en el caso de tratar con atributos numéricos:

$$de(x, d_i) = \sqrt{\sum_{j=1}^{m}(x_j - d_{ij})^2} \tag{1}$$

En el caso de tratar con atributos nominales o binarios, una de las más utilizadas es la distancia de Hamming:

$$dh(x, d_i) = \sum_{j=1}^{m}\delta(x_j, d_{ij}) \tag{2}$$

donde $\delta(x_j, d_{ij})$ toma el valor 0 si $x_j = d_{ij}$ y 1 en caso contrario.

El valor de la variable k es muy importante en la ejecución de este método, de modo que con valores distintos de k podemos obtener resultados también muy distintos. Este valor suele fijarse tras un proceso de pruebas con varias instancias. Se suele escoger un número impar o primo para minimizar la posibilidad de empates en el momento de decidir la clase de una nueva instancia. Aún así, cuando se produce un empate se debe decidir cómo clasificar la instancia. Algunas alternativas pueden ser, por ejemplo, no dar predicción o dar la clase más frecuente en el conjunto de aprendizaje de las clases que han generado el empate.

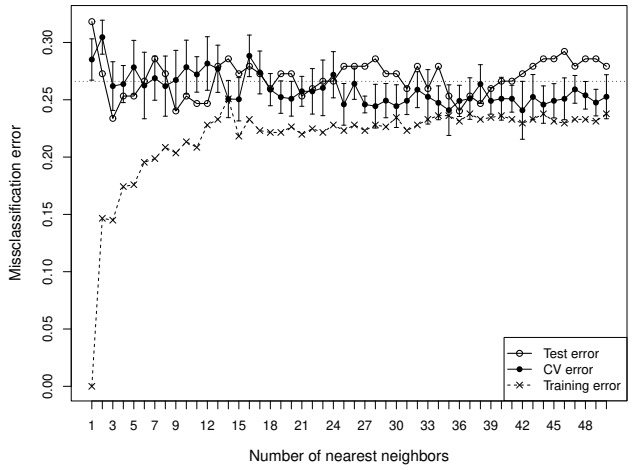
Otro factor importante que se debe considerar antes de aplicar el algoritmo k-NN es el rango u orden de los datos. Es importante expresar los distintos atributos en valores que sean "comparables". Por ejemplo, si modificamos un atributo que originalmente estaba expresado en kilogramos y lo transformamos en gramos, el resultado de las métricas de distancias pueden cambiar drásticamente. En este sentido, existen transformaciones, como por ejemplo la unidad tipificada (*standar score* o *z-score* en inglés) que resta a cada valor el valor medio del atributo y lo divide por su respectiva desviación estándar.

Merece la pena comentar que puede llegar a producirse el efecto del sobreentrenamiento del modelo (*overfitting*). El sobreentrenamiento provoca que el modelo que se construye, en lugar de describir las estructuras subyacentes del juego de datos, lo que acaba describiendo sea exclusivamente el juego de datos de entrenamiento no siendo extrapolable a otros juegos de datos.

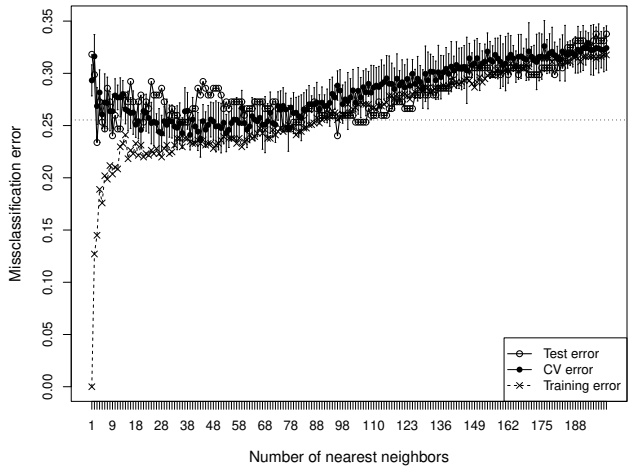
2.3. Ejemplo de selección empírica del valor de k

Veamos un ejemplo de cómo seleccionar el valor de la k utilizando el conjunto de datos PimaIndiansDiabetes[1]. Este conjunto de datos presenta 768 instancias y 9 atributos utilizados para determinar si un paciente tiene diabetes. El 35 % de las instancias pertenecen a la clase de pacientes positivos, es decir, que finalmente se les diagnosticó diabetes.

Figura 2. Evolución del error en función del parámetro k



(a) Valores de k en el rango $\{1, \ldots, 50\}$



(b) Valores de k en el rango $\{1, \ldots, 200\}$

La Figura 2 (a) muestra los resultados de un conjunto de pruebas con valores de $k = \{1, \ldots, 50\}$ (eje horizontal) utilizando el método *10-fold cross-validation*. La figura muestra tres tipos de errores (eje vertical), que se corresponden con el error del conjunto de test (*test error*), el error en las pruebas de validación cruzada (*CV error*) y el error en el conjunto de entrenamiento (*training error*). Como se puede apreciar, a partir de un valor de $k = 15$, el error de test y de validación cruzada no experimenta cambios importantes. Por lo tanto, un valor entre 18 y 26 parece adecuado. Es interesante notar que el error de entrenamiento es muy bajo para valores

[1] `https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes`

de $1 \leq k \leq 5$, pero considerablemente por encima de la media en el caso de considerar un conjunto de test o una estimación mediante validación cruzada.

Cuando el valor del parámetro k aumenta demasiado, aparte del coste computacional, estaremos considerando instancias próximas y no tan próximas para determinar la clase de un individuo concreto, lo que deriva en problemas de subentrenamiento (*underfitting*). La Figura 2 (b) muestra un proceso similar al visto en la Figura 2 (a), pero donde analizamos el comportamiento del algoritmo en un intervalo de $k = \{1, \ldots, 200\}$. Se puede apreciar claramente como el comportamiento del algoritmo se degrada de forma considerable para un valor de $k \geq 70$.

2.4. Resumen

Los principales puntos fuertes de este método de clasificación supervisado son la simplicidad y la robustez ante el ruido presente en los datos de entrenamiento. Además, el hecho de "crear" específicamente un modelo para procesos de predicción puede ser visto como una ventaja importante en ciertos problemas o dominios.

Por el contrario, algunas de las principales limitaciones de este método también están relacionadas con el hecho de que no se cree un modelo específico durante el entrenamiento. Esto provoca que cada nueva instancia deba ser comparada con los demás para encontrar los k vecinos más próximos e identificar la clase de la nueva instancia. Por lo tanto, el proceso de clasificación de cada nueva instancia no es trivial a nivel de coste computacional, y crece al aumentar el conjunto de entrenamiento etiquetado.

En este sentido, es muy importante escoger una implementación del algoritmo que permita una ejecución óptima de cálculo y memoria. Además, el rendimiento se puede ver seriamente afectado cuando se trata con conjuntos con muchos atributos, como por ejemplo en el caso de imágenes que pueden presentar cientos o miles de dimensiones, problema que se conoce como la "maldición de la dimensión" (*the curse of dimensionality*).

3. Máquinas de soporte vectorial

Las máquinas de soporte vectorial (en inglés, *Support Vector Machines* o SVM) es el nombre con el cual se conoce un algoritmo de aprendizaje supervisado capaz de resolver problemas de clasificación, tanto lineales como no lineales. Actualmente está considerado como uno de los algoritmos más potentes en reconocimiento de patrones.

Su eficiencia y los buenos resultados obtenidos en comparación con otros algoritmos han convertido esta técnica en la más utilizada en campos como el reconocimiento de textos y habla, predicción de series temporales y estudios sobre bases de datos de marketing, entre otros, pero también en otros campos como la secuenciación de proteínas y el diagnóstico de varios tipos de cáncer.

La gran aportación de Vapnik (Vapnik & Chervonenkis, 1974) radica en que construye un método que tiene por objetivo producir predicciones en las que se puede tener mucha confianza, en lugar de lo que se ha hecho tradicionalmente, que consiste en construir hipótesis que cometan pocos errores.

Lectura complementaria

V. N. Vapnik, A. Ya. Chervonenkis, "On the method of ordered risk minimization. I", Avtomat. i Telemekh., 1974, no. 8, 21?30; Autom. Remote Control, 35:8 (1974), pp. 1226-1235

La hipótesis tradicional se basa en lo que se conoce como minimización del riesgo empírico (*empirical risk minimization*) mientras que el enfoque de las SVM se basa en la minimización del riesgo estructural (*structural risk minimization*), de modo que lo que se busca es construir modelos que estructuralmente tengan poco riesgo de cometer errores ante clasificaciones futuras. El concepto de minimización del riesgo estructural fue introducido en 1974 por Vapnik y Chervonenkis (Vapnik & Chervonenkis, 1974).

Para entender mejor tanto la teoría de Vapnik y Chervonenkis aplicada a las SVM, vamos a simplificar y focalizarnos en el problema de la clasificación binaria, en la que a partir del conjunto de datos de entrenamiento se construye un hiperplano (separador lineal) capaz de dividir los puntos en dos grupos.

La idea detrás de las máquinas de soporte vectgorial es muy intuitiva. Si la frontera definida entre dos regiones con elementos de clases diferentes es compleja, en lugar de construir un clasificador complejo que reproduzca dicha frontera, lo que se intenta es "doblar" el espacio de datos en un espacio de mayor dimensionalidad de forma que con un único corte (es decir, un clasificador muy sencillo) se puedan separar fácilmente ambas regiones.

3.1. Definición de margen e hiperplano separador

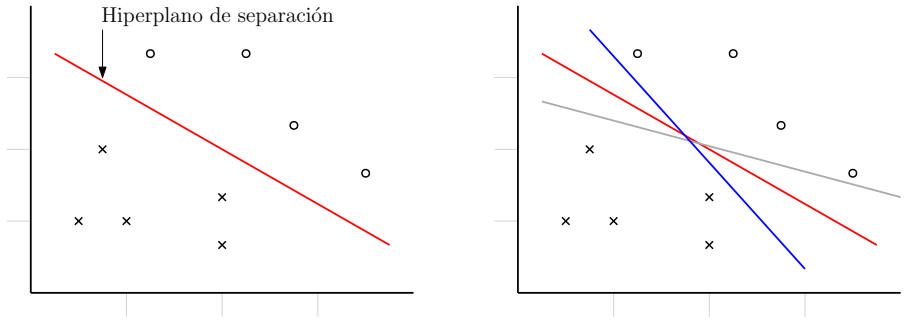
Definiremos el **margen** como la zona de separación entre los distintos grupos a clasificar. Esta zona de separación quedará delimitada a través de **hiperplanos**.

Uno de los factores diferenciadores de este algoritmo respecto de otros clasificadores es su gestión del concepto de margen. Las SVM buscan maximizar el margen entre los puntos pertenecientes a los distintos grupos a clasificar. Maximizar el margen implica que el hiperplano de decisión o separación esté diseñado de tal forma que el máximo número de futuros puntos queden bien clasificados. Por este motivo, como veremos más adelante, las SVM utilizan las técnicas de optimización cuadrática propuestas por el matemático italiano Giuseppe Luigi Lagrange.

El objetivo de las SVM es encontrar el hiperplano óptimo que maximiza el margen entre clases (variable objetivo) del juego de datos de entrenamiento.

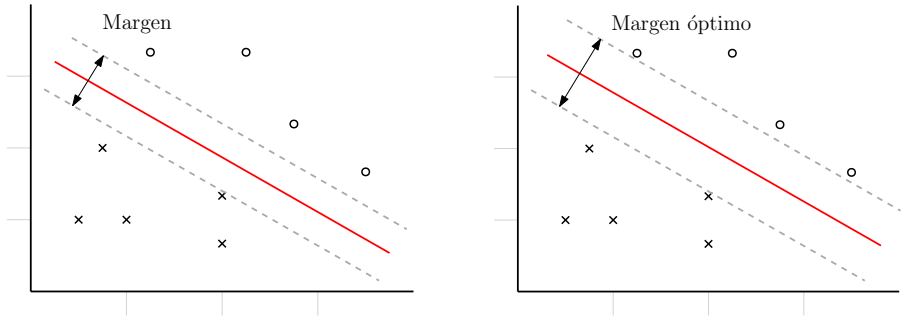
Veamos en la Figura 3 (a) un gráfico de dispersión del juego de datos de entrenamiento, donde vemos las dos clases presentes en el conjunto de datos.

Figura 3. Ejemplo de datos binarios para ilustrar el concepto de margen e hiperplano



(a) Conjunto de puntos e hiperplano de separación

(b) Hiperplanos de separación



(c) Hiperplano con margen

(d) Hiperplano con margen óptimo

A partir de la Figura 3 (a) podemos observar que a partir de un hiperplano podríamos separar las dos clases de datos de forma clara. Diremos que esta recta de separación es en realidad un **hiperplano de separación**.

Debemos tener en cuenta que el hecho de disponer de un hiperplano separador no nos garantiza, en absoluto, que éste sea el mejor de todos los posibles hiperplanos

separadores. Vemos en la Figura 3 (b) que en nuestro ejemplo podemos tener infinitos hiperplanos separadores, todos ellos perfectamente válidos para separar nuestro juego de datos de clientes entre aquellos que contratan y aquellos que no contratan el servicio de seguro.

Ante la pregunta de si hay o no un hiperplano separador mejor que otro, pensemos, por ejemplo, que aquellos hiperplanos que estén demasiado cerca de los puntos del gráfico, muy probablemente no serán demasiado buenos para clasificar nuevos puntos, ya que correremos mucho riesgo de acabar teniendo puntos en el lado no deseado del hiperplano. Por lo tanto, nuestra intuición nos dice que sería conveniente fijar un margen de seguridad capaz de establecer una distancia entre los puntos de cada clase de puntos a clasificar.

Como puede observarse en la Figura 3 (c), el margen es un "terreno de nadie", donde no deberíamos encontrar ningún punto del juego de datos de entrenamiento. Para construir el margen asociado a un hiperplano, procederemos a calcular la distancia entre el hiperplano y el punto más cercano a éste. Una vez tenemos esta distancia, la doblamos y éste será nuestro margen. De este modo, cuanto más cerca esté un hiperplano de los puntos, menor será su margen. Por contra, cuanto más alejado esté un hiperplano de los puntos, mayor será su margen y, en consecuencia, menor será su riesgo de clasificar incorrectamente nuevos puntos del juego de datos.

Así, el **hiperplano separador óptimo** se define como el hiperplano que tenga mayor margen posible, tal y como podemos ver en la Figura 3 (d). El objetivo de las SVM es encontrar el hiperplano separador óptimo que maximice el margen del juego de datos de entrenamiento.

3.2. Cálculo del margen y del hiperplano separador óptimo

En esta sección usaremos cálculo vectorial para poder operar con vectores y estudiar las propiedades de los mismos, que nos van a permitir obtener la ecuación del hiperplano separador con el margen óptimo.

3.2.1. La ecuación del hiperplano

La expresión más habitual para representar una recta en el plano es $y = ax + b$. Si generalizamos este caso a un espacio de más dimensiones, por cuestiones de practicidad se suele usar la expresión $w^T \cdot x = 0$, donde la equivalencia con la expresión anterior se muestra a continuación:

$$w = \begin{pmatrix} -b \\ -a \\ 1 \end{pmatrix}, x = \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} \tag{3}$$

A partir de la notación, es fácil operar y ver que estas dos expresiones son equivalentes:

$$w^T \cdot x = (-b)(1) + (-a)x + (1)y = y - ax - b \tag{4}$$

Aunque ambas ecuaciones son equivalentes, generalmente, es más cómodo trabajar con la expresión $w^T \cdot x$ por los siguientes motivos:

- Es más adecuada para más de dos dimensiones.
- El vector w siempre será perpendicular al hiperplano, por definición.

Un cálculo que se repite constantemente en las SVM es la distancia entre un punto x_0 y un hiperplano definido por su vector normal w. Este cálculo se efectúa mediante una proyección del vector definido entre el punto x_0 y el hiperplano en cuestión sobre el vector normal al hiperplano. De forma resumida:

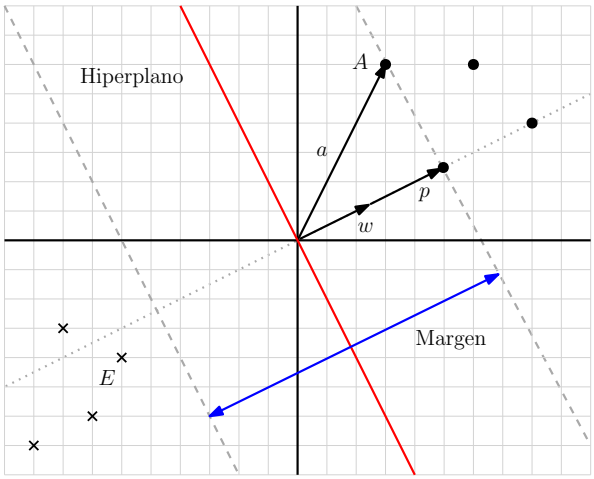
$$d = \frac{|x_0 \cdot w|}{\|w\|} \tag{5}$$

Entonces, si x_0 era el punto más cercano al hiperplano, podemos definir el margen como el doble de la distancia calculada (extendiendo dicha distancia a cada lado del hiperplano), dado que en ese margen no existe ningún otro punto.

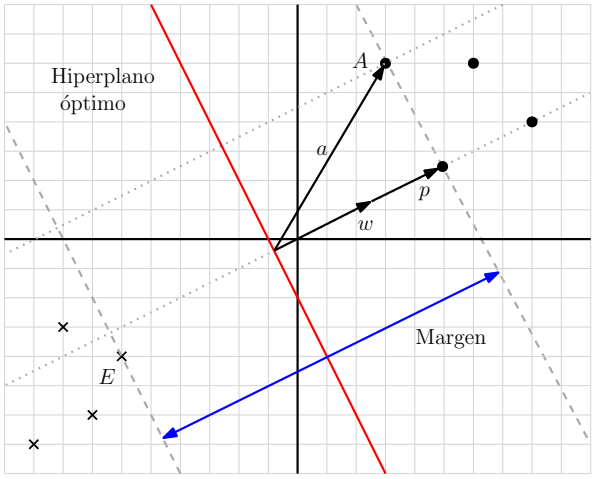
En la Figura 4 (a) podemos apreciar como el margen calculado para el punto A no es el óptimo, ya que podríamos conseguir un margen mayor hasta alcanzar el punto E, haciendo retroceder el hiperplano en cuestión. Sin embargo, en la Figura 4 (b) vemos como el margen alcanza los puntos A y E que marcan la frontera entre los dos grupos de puntos. Diremos entonces que el margen e hiperplano generados son los óptimos.

Más adelante veremos como el algoritmo SVM llama a los puntos A y E, **vectores**

Figura 4. Ejemplo de datos bidimensionales para ilustrar el cálculo del margen óptimo



(a) Hiperplano y margen para el punto A



(b) Hiperplano óptimo entre los puntos A y E

de soporte, es decir, vectores definidos por puntos que marcarán la frontera que nos servirá para calcular el margen y el hiperplano separador.

Obviamente, los conceptos hiperplano y margen están intrínsecamente relacionados, puesto que fijado un margen podemos obtener el hiperplano que pasa justo por su centro y al revés, fijado un hiperplano podemos obtener el margen a partir de la distancia de los puntos frontera con el hiperplano. Por lo tanto, podemos concluir que determinar el mayor margen posible es equivalente a encontrar el hiperplano óptimo.

El proceso de optimización se completa con el método de los **multiplicadores de Lagrange**, adecuado para encontrar máximos y mínimos de funciones continuas de múltiples variables y sujetas a restricciones. En dos dimensiones, los multiplicadores de Lagrange permiten resolver el problema de maximizar una función $f(x, y)$ sujeta a las restricciones $g(x, y) = 0$, con la única condición de que ambas tengan derivadas parciales continuas, es decir, que su forma sea "suave" en todas sus variables. De este modo, Lagrange optimizará la función $f(x, y) - \lambda \cdot g(x, y)$, donde λ es el coste de la restricción. Esto es generalizable a cualquier número de dimensiones.

En el caso de las SVM, el objetivo es encontrar aquel hiperplano que maximiza el margen (la distancia del punto más cercano a dicho hiperplano) y que satisface que todos los puntos de una clase están a un lado del hiperplano y todos los de la otra clase están al otro lado. Cuando esto es posible se habla de "hard-margin". Si esto no es posible (es decir, el conjunto de datos no es linealmente separable), se habla de "soft-margin" y es necesario añadir una condición más de forma que el hiperplano óptimo minimize el número de elementos de cada clase que se encuentran en el lado equivocado, teniendo en cuenta también su distancia.

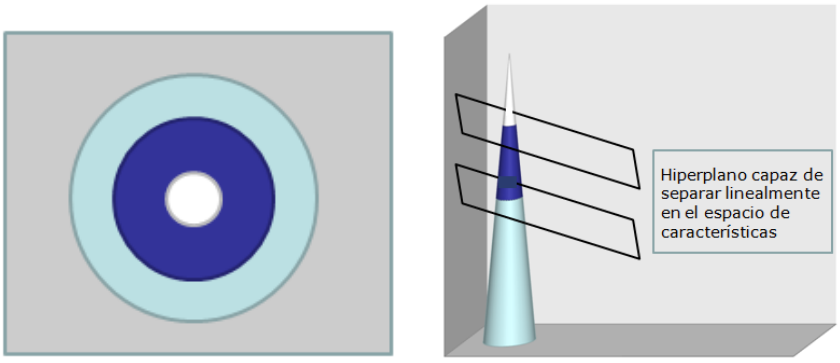
3.3. Funciones *kernel*

Estrictamente hablando, las SVM se ocupan tan solo de resolver problemas de clasificación lineal y se apoyan en las funciones *kernel* para transformar un problema no lineal en el espacio original X en un problema lineal en el espacio transformado F. Las funciones *kernel* son las que "doblan" el espacio de entrada para poder realizar cortes complejos con un simple hiperplano.

3.3.1. ¿Cómo pueden ayudarnos las funciones *kernel*?

La Figura 5 (a) nos muestra un problema de clasificación con tres zonas de distintos colores. Claramente se trata de un problema de clasificación no lineal puesto que visualmente observamos cómo solo dos circunferencias concéntricas son capaces de realizar una correcta clasificación.

Figura 5. Ejemplo de clasificación y función *kernel* radial



(a) Clasificación radial (b) Función *kernel* radial

Supongamos que disponemos de una función *kernel* capaz de transportar la figura anterior a un espacio de tres dimensiones, tal y como podemos ver en la Figura 5 (b). De hecho, podemos pensar la Figura 5 (a) como un plano en picado de un cono dibujado, tal y como vemos en la Figura 5 (b).

En el espacio transformado por la función *kernel* (también llamado espacio de características) sí que podemos separar los tres colores por planos, es decir, tenemos

un problema de clasificación lineal. En este nuevo espacio podríamos identificar el margen óptimo de separación para clasificar las tres zonas de colores diferentes.

3.3.2. Función *kernel*

Una función *kernel* es aquella que a cada par de vectores de un espacio origen $X \subseteq \mathbb{R}^n$, asigna un número real, cumpliendo con las propiedades del producto escalar.

$$k : X \times X \to \mathbb{R}, \text{ donde } X \subseteq \mathbb{R}^n \tag{6}$$

Las propiedades que debe cumplir una función *kernel* son:

- La propiedad distributiva.
- La propiedad conmutativa.
- La propiedad semidefinida positiva.

En términos algebraicos el producto escalar es la suma de los productos de los correspondientes componentes de cada vector. Así mismo, en términos geométricos podemos pensar el producto escalar como la proyección de un vector sobre otro, tal y como hemos estudiado en la sección anterior.

En definitiva, una función *kernel* no es más que una versión "modificada" de la definición clásica de producto escalar. Este tipo de funciones reciben el nombre de funciones *kernel* porque se construyen a partir de una asignación de pesos a los distintos componentes de cada vector, tal y como podemos apreciar en la ecuación 7.

3.3.3. Algoritmo SVM

Para entenderlo mejor, veamos un caso de clasificación binaria, es decir, cada punto de nuestro espacio de características, $x_i \in X$, estará asociado a una clase binaria, i.e. $c_i \in \{-1, 1\}$.

Podríamos pensarlo como la medida de similitud que nos permita determinar si un nuevo punto se encuentra a la derecha o a la izquierda de un hiperplano de separación.

En este caso, nuestra función de clasificación podría consistir en calcular la suma de distancias alteradas por un peso w, como se muestra a continuación:

$$h(z) = signo\Big(\sum_{i=1}^{n} w_i \cdot c_i \cdot k(x_i, z)\Big) \tag{7}$$

donde:

- n es el número de entradas del juego de datos de entrenamiento.
- z es el nuevo punto a clasificar.
- $h(z) \in \{-1, 1\}$ es la función de clasificación.
- $k : X \times X \to \mathbb{R}$ es la función *kernel* que mide la similitud entre puntos (producto escalar, distancia, etc).
- El conjunto de pares $\{(x_i, c_i)\}_{i=1}^{n}$ son los puntos etiquetados del juego de datos. Es decir, constituyen el juego de datos de entrenamiento, donde $c_i \in \{-1, 1\}$ es la etiqueta o clase del punto x_i.
- $w_i \in \mathbb{R}$ son los pesos que el algoritmo ha determinado para el juego de datos de entrenamiento.
- *signo*() es la función que nos devuelve el signo de un número, i.e. + o −.

El trabajo del algoritmo SVM será, precisamente, determinar los valores w óptimos en términos de maximización del margen, con el objetivo de encontrar el mejor hiperplano de separación.

3.3.4. El método *kernel*

En ocasiones puede ser interesante añadir a la Ecuación 6 una función adicional de transformación del espacio original X en un nuevo espacio intermedio V:

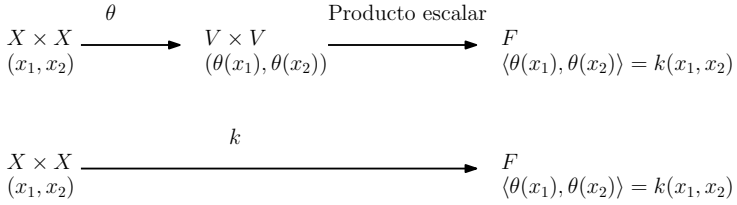
$$\phi : X \to V \ donde \ X \subseteq \mathbb{R}^n;\ V \subseteq \mathbb{R}^m \tag{8}$$

con $m > n$, de tal modo que:

$$k : X \times X \to V \times V \to F \text{ donde } X \subseteq \mathbb{R}^n;\ V \subseteq \mathbb{R}^m;\ F \subseteq \mathbb{R} \tag{9}$$

La Figura 6 muestra el detalle del proceso comentado hasta ahora, donde nuevamente la operación $<,>$ deberá cumplir la condición de ser producto escalar.

Figura 6. Función *kernel* como producto escalar



De este modo, nuestra función *kernel* $k(x_1, x_2)$ podría pensarse como un producto escalar $< \phi(x_1), \phi(x_2) >$, mucho más fácil de operar ya que podemos aplicar las propiedades algebraicas y geométricas del producto escalar.

Esta transformación del espacio de características, que consiste en hacer que nuestra función *kernel* opere en un espacio de más dimensiones, podréis encontrarla en la literatura inglesa como *kernel trick* o método *kernel*.

Gracias a este tipo de transformaciones, conseguimos que juegos de datos que en el espacio $\mathbb{R}^n$ no son linealmente separables, sí que lo sean en un espacio de mayor dimensionalidad, i.e. $\mathbb{R}^m$ donde $m > n$.

La principal potencia de este método es que en realidad no necesitamos trabajar explícitamente en el espacio $\mathbb{R}^m$ ya que simplemente necesitamos operar el producto escalar del espacio transformado. A continuación veremos algunos ejemplos de transformaciones.

Transformación cuadrática en un espacio de más dimensiones

La transformación:

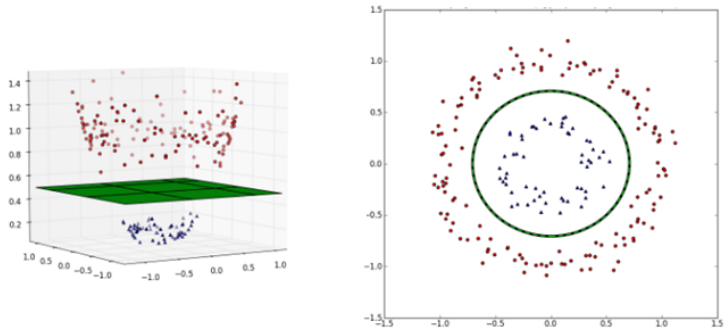
$$[u_1, u_2] = [u_1, u_2, u_1^2 + u_2^2] \tag{10}$$

es la función $\phi : \Re^2 \to \Re^3$ que usaremos para una función *kernel* lineal (producto escalar clásico), de modo que:

$$k(u,v) = u^T \cdot v =< \phi(u_1,u_2), \phi(v_1,v_2) > \\ =< (u_1,u_2,u_1^2+u_2^2), (v_1,v_2,v_1^2+v_2^2) > \tag{11}$$

En la Figura 7 vemos muy gráficamente el beneficio de nuestra transformación, donde añadiendo adecuadamente una dimensión más, conseguimos trabajar con un juego de datos que sí es linealmente separable.

Figura 7. Ejemplo de *kernel trick*



Transformación cuadrática manteniendo las dimensiones

Imaginemos una distribución concéntrica como la que apreciamos en la Figura 8 (a).

Lo que quisiéramos es una función que transforme el espacio original en un espacio de características en el que la distribución de puntos sea linealmente separable, es decir, que sea separable a través de un hiperplano.

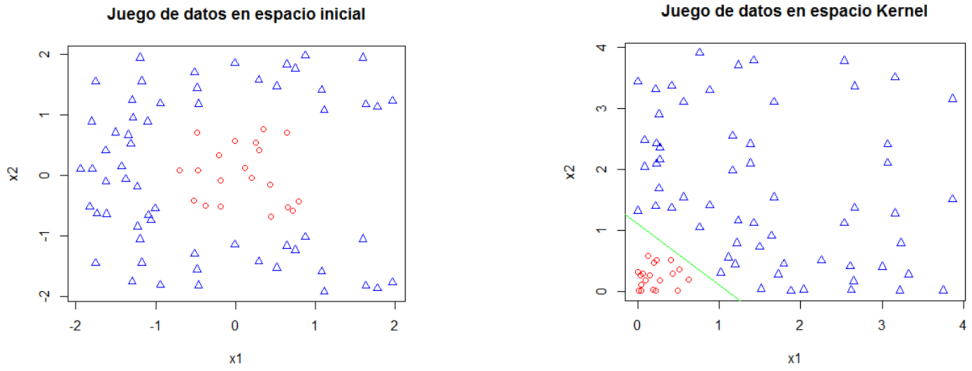
Una función *kernel* que nos ayudará a realizar esta transformación es la función cuadrática que simplemente consiste en considerar el cuadrado de cada uno de los dos componentes de cada punto, manteniendo el número de dimensiones. El espacio de características que conseguiríamos sería el que apreciamos en la Figura 8 (b).

La función *kernel* de nuestro ejemplo podría tener dos expresiones equivalentes:

$$k(u,v) = u_1^2 v_1^2 + u_2^2 v_2^2 \tag{12}$$

donde $u, v \in X \subseteq \mathbb{R}^2$, y

Figura 8. Ejemplo de transformación cuadrática manteniendo las dimensiones



(a) Distribución de puntos en el espacio original (b) Distribución de puntos en el espacio de características

$$k(u,v) =< (u_1^2, u_2^2), (v_1^2, v_2^2) >=< \phi(u), \phi(v) > \tag{13}$$

donde $u, v \in X \subseteq \mathbb{R}^2$.

Probablemente, las matemáticas nos ayudan más si lo pensamos como la Ecuación 13, dado que a partir de la definición de producto escalar, podemos encontrar el margen óptimo de separación.

Transformación polinomial

La transformación:

$$[u_1, u_2] = [u_1^2, u_1 \cdot u_2\sqrt{2}, u_1 \cdot \sqrt{2b}, u_2 \cdot \sqrt{2b}, b] \tag{14}$$

es la función $\phi : \mathbb{R}^2 \to \mathbb{R}^5$ que usaremos para una función *kernel* polinomial

$$k(u,v) = (u^T \cdot v + b)^2 \tag{15}$$

Esta técnica es muy eficiente en términos de computación porque en realidad siempre estamos computando un producto escalar. En un espacio con más dimensiones, pero

al final siempre computamos un producto escalar para medir la distancia de nuestro punto o vector al hiperplano de separación.

3.4. Tipos de funciones *kernel*

Una de las tareas que realiza el algoritmo SVM es identificar los **vectores de soporte**, es decir, aquellos vectores que marcaran la trayectoria del hiperplano separador.

Los vectores de soporte marcan la frontera. Sería como una línea de seguridad que una vez sobrepasada deja de ofrecer garantías de fiabilidad en la predicción, de modo que a la hora de valorar un modelo predictivo basado en las SVM podríamos plantearnos el cuantificar, por ejemplo, qué porcentaje de predicciones se encuentran alejadas de los vectores de soporte y qué porcentaje de predicciones se encuentran en sus aledaños.

Veremos a continuación que cada estrategia de clasificación (lineal, polinomial, radial y sigmoidal) tiene su propia función *kernel*. Tal y como menciona el autor Jean Philippe Vert [**?**] la idea que hay detrás de todas ellas es un concepto equivalente al del producto escalar que persigue medir el grado de similitud entre dos puntos (vistos como vectores) de nuestro juego de datos.

3.4.1. Kernel lineal

La función *kernel* utilizada para la SVM de clasificación lineal es:

$$k(u, v) = u^T \cdot v \tag{16}$$

donde $u, v \in X$.

Observamos que simplemente se trata del producto escalar de dos vectores y no supone ninguna transformación del espacio de características. En este caso la función *kernel* responde enteramente a lo que nuestra intuición entiendo por producto escalar. La Figura 9 (a) muestra un ejemplo de la superficie de predicción creada por un modelo lineal.

3.4.2. Kernel polinomial

La función *kernel* polinomial es:

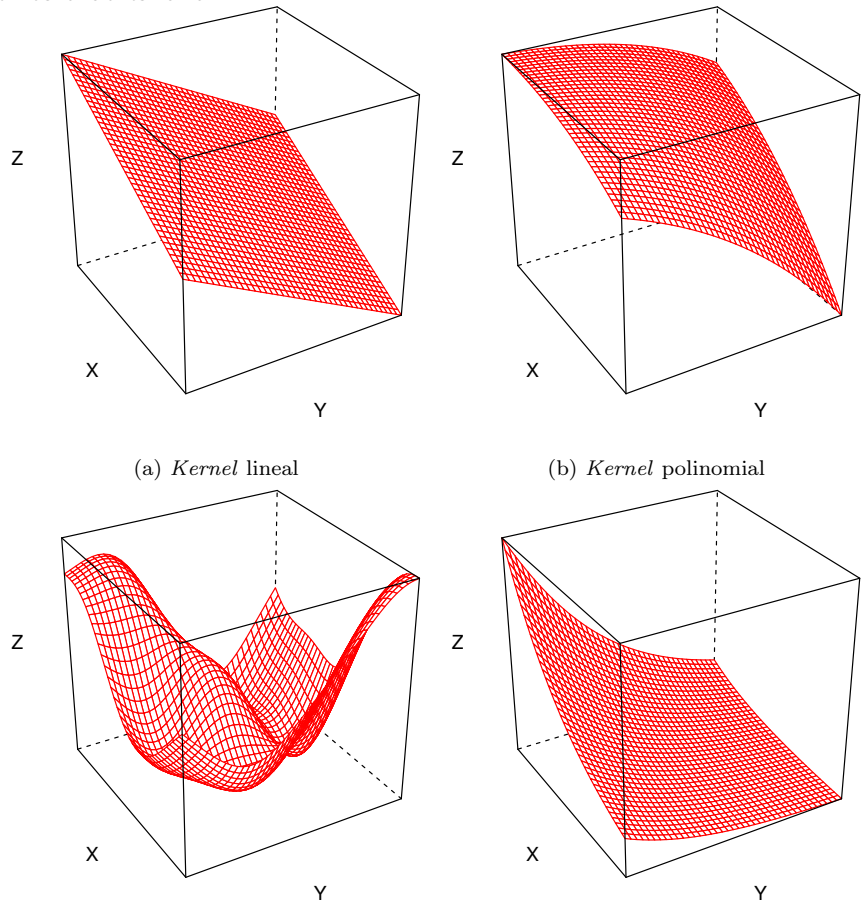
$$k(u, v) = (\gamma u^T \cdot v + b)^p \tag{17}$$

donde $u, v \in X$ y $\gamma > 0$, $b \geq 0$ y $p > 0$ son parámetros.

Podríamos pensar el *kernel* polinomial como una primera generalización del concepto de producto escalar, en el que tenemos la opción de trabajar en un espacio transformado de más dimensiones que el espacio original.

Los parámetros γ, b, p nos permiten modelar en cierto grado la morfología de la transformación del espacio original. El calibraje de estos parámetros y la influencia que cada uno de ellos ejerce sobre la transformación del juego de datos, forman parte del estudio necesario para poder usar de forma eficiente esta función *kernel*. La Figura 9 (b) muestra un ejemplo de la superficie de predicción creada por un modelo polinomial.

Figura 9. Ejemplo de las distintas formas en el espacio transformado que podemos gestionar con las funciones *kernel*



(a) *Kernel* lineal

(b) *Kernel* polinomial

(c) *Kernel* radial

(d) *Kernel* sigmoidal

3.4.3. Kernel radial

La función *kernel* radial es:

$$k(u, v) = e^{-\gamma||u-v||^2} \tag{18}$$

donde $u, v \in X$ y $\gamma > 0$ es parámetro.

También conocido como *kernel* de Gauss, generaliza el concepto de distancia entre vectores en lugar de su producto escalar. Está especialmente indicada para modelar relaciones complejas y no lineales, como podemos ver en el ejemplo de superficie predicción de la Figura 9 (c). Esta función es muy dependiente del parámetro γ, de modo que será importante tenerlo en cuenta en aspectos como el posible sobreentrenamiento del juego de datos.

3.4.4. Kernel sigmoidal

La función *kernel* sigmoidal es:

$$k(u, v) = \tanh(\gamma u^T \cdot v + b) \tag{19}$$

donde $u, v \in X$ y $\gamma > 0$, $b \geq 0$ son parámetros.

Se trata de la tangente hiperbólica del producto escalar en el espacio original, de modo que sin duda se trata de otro caso de generalización del producto escalar.

Es capaz de modelar relaciones complejas similares al *kernel* radial. La tangente hiperbólica es una función muy utilizada como función de activación en las redes neuronales y por este motivo también tiene un papel importante en el mundo de las SVM. La Figura 9 (d) muestra un ejemplo de la superficie de predicción creada por un modelo sigmoidal.

Sin embargo, tiene el riesgo que para ciertos valores de sus parámetros γ, b puede llegar a representar una función *kernel* no válida, es decir, que no cumpla con las condiciones que se le exigen a una función *kernel* para ser considerada como tal.

3.5. Resumen

La apuesta de las máquinas de soporte vectorial por maximizar las opciones de clasificar correctamente las nuevas entradas del juego de datos, convierten este algoritmo en un clasificador muy robusto cuando los datos son claramente separables (mediante un hiperplano), sin embargo, cuando hay ruido, sobre todo en la franja de separación, puede dar algunos problemas.

Las SVM son muy efectivas en juegos de datos con muchas dimensiones o atributos, sacando provecho de sus propiedades geométricas. Además, gracias al recurso de los vectores de soporte que funcionan como puntos de referencia, se consigue una gran eficiencia computacional.

Actualmente, las SVM son un algoritmo muy utilizado en tareas de clasificación de documentos o también en el ámbito de la salud, como por ejemplo en la prevención y detección de enfermedades de diagnóstico complejo.

En definitiva y a pesar de que también pueden usarse para tareas de regresión, las máquinas de soporte vectorial son un algoritmo muy eficiente y ampliamente usado en tareas de clasificación. El uso que hacen de las funciones *kernel* y su visión puramente vectorial del problema de clasificación le confieren una potencia matemática que explica claramente su buen rendimiento ante otras tipologías de algoritmos.

El principal problema de las SVM es la dificultad para entender el hiperplano de corte en el espacio de mayor dimension donde se realiza la separación de elementos de clases diferentes, dado que puede no resultar trivial entender el “pliegue”del espacio de entrada que realiza la función *kernel*, funcionando entonces como un modelo de caja negra.

4. Árboles de decisión

Los árboles de decisión son uno de los modelos de minería de datos más comunes y estudiados, y no precisamente por su capacidad predictiva, superada generalmente por otros modelos más complejos, sino por su alta capacidad explicativa y la facilidad para interpretar el modelo generado. Por ejemplo, son ampliamente utilizados en sectores como el bancario y las compañías aseguradoras para tomar decisiones respecto a la concesión de créditos o el cálculo de las pólizas, respectivamente, dado que permiten determinar qué características relativas a los usuarios son las que comportan mayor o menor riesgo, siendo posible segmentar a los usuarios en función de dichas características.

Lectura complementaria

Leo Breiman, Jerome Friedman, Charles J. Stone, R.A. Olshen, (1984). "Classification and Regression Trees", Chapman and Hall/CRC

Como modelos supervisados que son, a los árboles de decisión que clasifican los datos del conjunto de entrada en función de una variable clasificadora categórica (es decir, que toma un conjunto finito de valores) se les llama árboles de clasificación. Si la variable clasificadora es continua, hablaríamos de árboles de regresión. Ambos tipos de árboles de decisión dan el nombre común CART (*Classification and Regression Trees*) según el trabajo original descrito por Breiman, Friedman, Stone y Olhsen en 1984 (Breiman *et al.*, 1984).

Básicamente, los árboles de decisión son un modelo de minería de datos que intenta subdividir el espacio de datos de entrada para generar regiones disjuntas, de forma que todos los elementos que pertenezcan a una misma región sean de la misma clase, la cual es utilizada como representante o clase de dicha región. Si una región contiene datos de diferentes clases es subdividida en regiones más pequeñas siguiendo el mismo criterio, hasta particionar todo el espacio de entrada en regiones disjuntas que solamente contienen elementos de una misma clase. Así, un árbol de decisión se llama completo o puro si es posible generar una partición del espacio donde cada subregión sólo contenga elementos de una misma clase. Esto siempre será posible, excepto si en el conjunto de datos de entrada existen elementos idénticos etiquetados con clases diferentes. En este caso será necesario decidir si se trata de *outliers* o bien se trata de casos que deben ser tratados de forma separada.

Respecto a su estructura, un árbol de decisión consta de nodos hoja o terminales, que representan regiones etiquetadas o clasificadas de acuerdo a una clase, y nodos internos o *splits* que representan condiciones que permiten decidir a qué subregión va cada elemento que llega a dicho nodo, es decir, la partición realizada. Cuando se presenta un nuevo dato a un árbol de decisión, se empieza por el nodo raíz que contiene una condición la cual determina por qué rama del árbol debe descenderse (es decir, a qué subregión pertenece el dato) hasta alcanzar la siguiente condición o bien una hoja terminal, en cuyo caso se determina qué clase corresponde al nuevo dato.

Contrariamente a la idea intuitiva ligada al concepto de “árbol”, la raíz es el elemento superior y las hojas se sitúan en posiciones más profundas. Así, la profundidad máxima de un árbol de decisión es el máximo número de condiciones que es necesario resolver para llegar a una hoja, siendo posible medir también la profundidad media, ponderando la profundidad de cada hoja con respecto al número de elementos del conjunto de entrenamiento que contiene.

Por lo tanto, un árbol de decisión es una secuencia de condiciones que son interrogadas con respecto a los datos de entrada, tomando una decisión parcial que lleva hacia una rama u otra, repitiendo este proceso hasta llegar a una hoja donde se toma una decisión final. Por ejemplo, en el hundimiento del Titanic, mostrado en la figura 10, de las 2201 personas a bordo solamente sobrevivieron 711 (un 32.3 %). Si se tuviera que tomar una decisión sobre todos los embarcados, esta sería "No sobrevivieron", dado que es correcta en el 67.7 % de los casos. Pero si analizamos la supervivencia con respecto a otras variables disponibles, la primera condición es el sexo, dado que las mujeres a bordo del Titanic (344 de 470, un 73.2 %) sobrevivieron en mayor proporción que los hombres (367 de 1731, un 21.2 %). Esto genera una partición del conjunto de entrada en dos conjuntos disjuntos; tras la primera condición (el nodo raíz), por una rama del árbol solo quedarán hombres y por la otra mujeres. La decisión con respecto a las mujeres sería "Sobrevivieron", dado que lo hicieron en un 73.2 %. En cambio, en el caso de los hombres, la decisión sería "No sobrevivieron", la cual sería cierta en un 78.8 % de los casos. Se puede observar que la precisión de la predicción ha aumentado para ambas hojas con respecto a la primera predicción. Esto siempre sucede al menos para una hoja, dado que el procedimiento seguido intenta mejorar la predicción anterior. De la misma manera, en el caso de los hombres, la siguiente condición es la edad, dado que los niños sobrevivieron en una proporción mayor que los adultos. En cambio, en el caso de las mujeres, la siguiente condición es la clase en la cual viajaban, dado que las mujeres que lo hacían en primera clase sobrevivieron más que las que lo hacían en segunda y mucho más que las que lo hacían en tercera clase. De hecho, las mujeres de tercera clase tuvieron una tasa de supervivencia inferior al 50 %, mientras que el resto sobrevivió en más de un 90 %. Nótese que cada rama puede contener una secuencia de condiciones diferentes, lo cual aporta información sobre el problema a resolver. De hecho, este árbol de decisión representa razonablemente bien la típica frase “las mujeres (no pobres) y los niños primero” usada en catástrofes de este tipo.

La Figura 11 muestra un ejemplo de árbol de decisión y una representación de la partición del espacio que genera, intentando separar elementos de dos clases diferentes en un espacio de datos de entrada de dos dimensiones. En este caso, la profundidad máxima del árbol es 2. La primera condición (C1) crea dos regiones, una de las cuales (R1) solamente contiene elementos de una clase, por lo que no es necesario continuar dividiéndola. La otra región es dividida de acuerdo a una segunda condición (C2) creando dos nuevas regiones (R2 y R3). La región R2 es también pura y no es necesario dividirla de nuevo, mientras que la región R3 aún contiene elementos de las dos clases, por lo que el algoritmo debería ejecutarse hasta conseguir un árbol completo.

Figura 10. Supervivencia de los embarcados en el Titanic

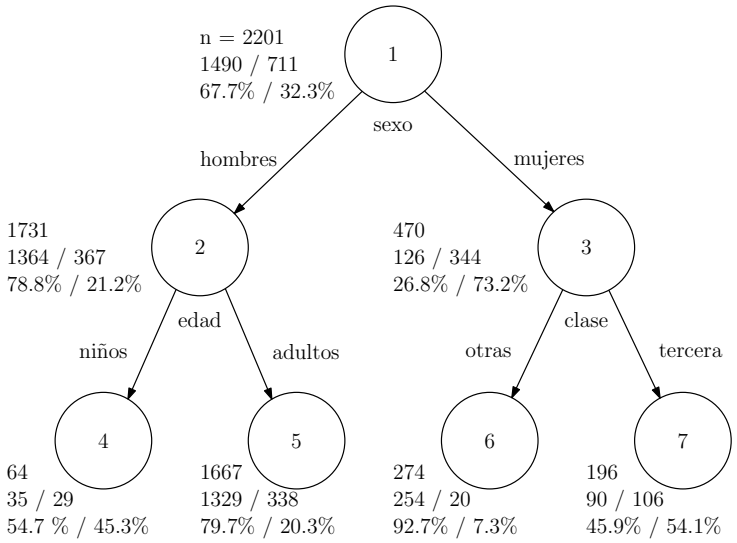
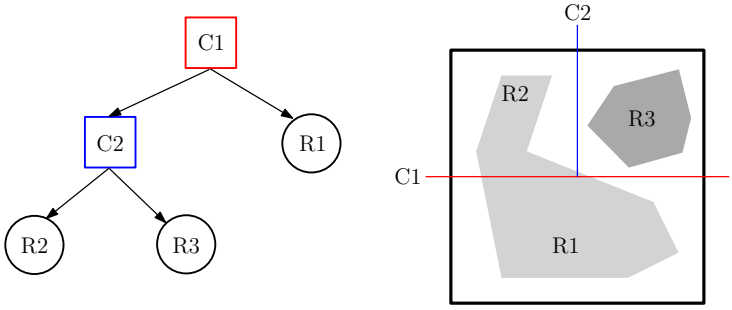


Figura 11. Ejemplo de árbol de decisión y la partición del espacio que genera



El algoritmo genérico para construir un árbol de decisión utiliza los siguientes criterios:

- C_p o criterio de parada, determina en qué momento se deja de seguir seleccionando nodos para ser subdivididos. El más habitual es generar un árbol completo, lo cual quiere decir que no quedan nodos que particionar.

- C_s o criterio de selección, determina qué nodo es seleccionado para ser particionado en dos o más subnodos. Existen diferentes criterios de selección, pero si se generan árboles de decisión completos el criterio utilizado no es relevante.

- C_c o criterio de clasificación, determina qué clase se asigna a un nodo hoja. Normalmente se trata de la clase que minimiza el error de clasificación o el valor numérico que minimiza el error de regresión.

- C_d o criterio de partición, determina cómo se particiona un nodo en dos o más subnodos. Normalmente los árboles de decisión son binarios (es decir, $P = 2$), indicando que cada región se subdivide en dos regiones disjuntas. Esto simplifica el criterio de partición y mejora la interpretabilidad del árbol a costa de generar árboles más profundos.

El algoritmo genérico para construir un árbol de decisión, o lo que es equivalente, una partición del espacio de entrada es el siguiente:

Algoritmo 2 Pseudocódigo para construir un árbol de decisión

Require: Conjunto de datos a clasificar D

$T = \{D\}$ // El árbol inicial es un solo nodo hoja
Etiquetar T de acuerdo a C_c
$p = \{T\}$ // Lista de nodos pendientes (hojas)
while no se deba parar según C_p **do**
  Seleccionar un nodo q de acuerdo a C_s
  if es posible particionar q según C_d **then**
    Particionar q en $q_1, \ldots, q_P$
    Etiquetar $q_1, \ldots, q_P$ según C_c
    Añadir $q_1, \ldots, q_P$ a p
    Substituir q en T por un nodo interno
  end if
  Eliminar q de p
end while
  return T

Los dos criterios más importantes son los de clasificación (C_c) y el de partición (C_d), dado que determinan el árbol construido a partir del conjunto de datos de entrada, especialmente cuando se construyen árboles completos.

4.1. Detalles del método

Como ya se ha comentado, el árbol de decisión construido depende de los cuatro criterios anteriormente mencionados: parada, selección, clasificación y partición. Los cuatro criterios están relacionados entre sí y pueden ser establecidos independientemente.

4.1.1. El criterio de parada

El criterio de parada determina cuándo se dejan de subdividir nodos para decidir que ya se ha construido un árbol suficientemente adecuado para los datos de entrada a clasificar. Este criterio suele usar aspectos relativos a la estructura del árbol y a su rendimiento como clasificador. Por ejemplo, puede decidirse parar cuando el árbol alcanza una cierta profundidad máxima o bien cuando la profundidad media (ponderada para todas las hojas) alcanza cierto valor. Otra posibilidad es medir el error cometido para los datos de entrada y parar cuando dicho error está por debajo de un cierto nivel. Obviamente, si no hay ninguna hoja que pueda ser particionada, el algoritmo también se detiene. En este sentido, si el objetivo es construir un árbol completo, el criterio de parada es simplemente no parar mientras exista alguna hoja que pueda ser particionada.

4.1.2. El criterio de selección

El criterio de selección determina qué nodo hoja es escogido para ser particionado. Obviamente, una hoja que solamente contenga elementos de una misma clase no debe ser particionada, ya que no mejora la capacidad predictiva del árbol, así que este criterio combina aspectos relativos a la impureza de las hojas, combinando el error cometido en dicha hoja y otras características como el número de elementos de conjunto de entrada que contiene o su profundidad en el árbol. Si el objetivo es construir un árbol completo, todas las hojas impuras deberán ser particionados, sea en el orden que sea, por lo que el criterio puede ser tan sencillo como seleccionar el siguiente nodo en la lista de nodos pendientes. No obstante, el criterio habitual es seleccionar el nodo más impuro, es decir, aquel que contiene una mayor mezcla de elementos de diferentes clases. Habitualmente esto puede medirse mediante la entropía, que mide el grado de desorden de una distribución de elementos de k clases diferentes:

$$H = \sum_{i=1}^{k} p_i \log p_i \tag{20}$$

La entropía es cero si todos los elementos son de una misma clase c, ya que entonces $p_c = 1$ y el resto de $p_i = 0$ $(\forall i \neq c)$ (se asume que $0 \log 0 = 0$). Cada p_i se calcula como el número de elementos de la clase i presentes en la hoja t dividido por el total de elementos en dicha hoja n_t, de la manera siguiente:

$$p_i(t) = n_i(t)/n_t \tag{21}$$

4.1.3. El criterio de clasificación

El criterio de clasificación determina qué clase se asigna a una región u hoja. Este criterio determina el error cometido en aquella hoja y también el error global que comete el árbol de decisión. Obviamente, si una hoja solamente contiene elementos de una clase, dicha clase es la elegida como representante de la región, minimizando el error cometido en la hoja, el cual es cero. En el caso de que en una hoja existan elementos de diferentes clases, se escogerá aquella clase que minimiza el error cometido, normalmente la más poblada. En el improbable caso de empate entre diferentes clases, puede tomarse una al azar. En general, la clase escogida es aquella que satisface la siguiente ecuación:

$$c(t) = \arg\min_{j} \sum_{i=1}^{k} p_i(t)\, C_{i,j} \tag{22}$$

Es decir, se trata de escoger aquella clase que minimiza el error cometido, teniendo en cuenta el posible diferente coste $C_{i,j}$ de cometer un error al escoger una clase i en lugar de la otra j, es decir, con costes asimétricos ($C_{i,j} \neq C_{j,i}$). Obviamente, $C_{i,i} = 0\ \forall i$. Por ejemplo, en un árbol de decisión que intenta determinar mediante la resonancia magnética de una muestra de tejido si ésta se trata de un tumor o no, es mejor cometer un error y obtener falsos positivos (es decir, indicar que hay un tumor cuando no es así) que no falsos negativos (es decir, indicar que no se trata de un tumor cuando realmente sí lo es).

El error total cometido por un árbol T es la suma ponderada del error cometido en cada hoja, teniendo en cuenta el tamaño de la región definido por cada hoja, estimado a través del número de elementos que la componen:

$$e(T) = \frac{1}{n} \sum_{t \in T} n_t \sum_{i=1}^{k} p_i(t)\, C_{i,c(t)} \tag{23}$$

que es equivalente a:

$$e(T) = \frac{1}{n} \sum_{t \in T} \sum_{i=1}^{k} n_i(t)\, C_{i,c(t)} \tag{24}$$

4.1.4. El criterio de partición

El criterio de partición determina cómo se divide una hoja en dos o más regiones. De hecho, el número de regiones y del criterio utilizado determinan qué tipo de árbol de decisión se está construyendo, dado que existen diferentes algoritmos que intentan aprovechar la naturaleza de los datos de entrada. Lo habitual es trabajar con árboles binarios, donde la región a dividir se particiona en dos regiones disjuntas. Se puede pensar como si un hiperplano cortara el espacio de entrada en dos. En el ejemplo de la Figura 11, cada una de las condiciones C1 y C2 representa este hiperplano.

El criterio de partición combina, de hecho, dos criterios diferentes: uno determina cómo se construye el hiperplano y el otro determina en qué posición del espacio se

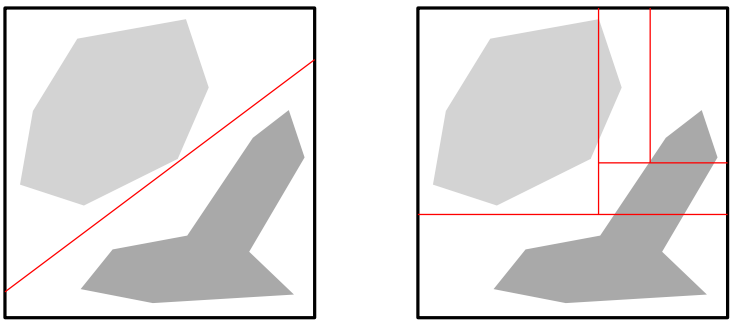
sitúa el hiperplano para realizar la partición.

En el primer caso, lo más habitual es utilizar hiperplanos ortogonales a cada uno de los ejes (variables) del espacio de entrada. Es decir, cada hiperplano es, en realidad, una condición que particiona los elementos de una hoja en dos subconjuntos en función de un valor dado. Es decir, en cada nodo interno sólo se utiliza una variable para decidir la clasificación de un elemento. Otra opción es buscar un hiperplano oblícuo el cual seguramente mejorará los resultados obtenidos con el método anterior, aunque la obtención de dicho hiperplano puede ser muy costosa, complicará su interpretación posterior y, además, puede provocar problemas de mala generalización para datos nunca vistos. En el caso mostrado en la Figura 12, un único hiperplano oblícuo puede separar perfectamente las dos clases presentes en el conjunto de datos, mientras que si se usaran hiperplanos ortogonales serían necesarios al menos cuatro, intentando reproducir el contorno de cada una de las regiones. El lector interesado en la construcción de hiperplanos oblícuos puede consultar el trabajo de Murthy *et al.*, 1994.

Lectura complementaria

Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. (1994). "A system for induction of oblique decision trees". J. Artif. Int. Res. 2, 1 (August 1994), 1-32. http://dl.acm.org/citation.cfm?id=162282

Figura 12. Ejemplo de hiperplano oblícuo



En el segundo caso, determinar la posición del hiperplano o corte para todos los hiperplanos posibles, lo que se hace es medir como mejoraría el árbol si se aplicara la partición definida por cada hiperplano, maximizando o minimizando algún criterio. Una opción que parece *a priori* interesante es utilizar el hiperplano que maximiza la reducción del error cometido por el árbol de decisión, de la forma siguiente. Sea T el árbol antes de realizar la partición de una hoja t y sea T_h el árbol resultante de particionar t en $t_{h\leq}$ y $t_{h>}$ mediante el hiperlano h. Se escogería, por lo tanto, el hiperplano tal que:

$$h = \arg\max_h \left(e(T) - e(T_h)\right) \tag{25}$$

No obstante, el uso del error de clasificación para los criterios de selección y partición está desaconsejado, siendo mejor usar otros criterios relacionados con la impureza del árbol, como la anteriormente mencionada entropía. El criterio más habitual es el llamado Índice de Gini o I_G, definido a partir de la probabilidad $p_i(t)$ de cada clase i en una hoja t como:

$$I_G(t) = \sum_{i=1}^{k} p_i(t)(1 - p_i(t)) \tag{26}$$

Entonces, para todos los hiperplanos posibles h que particionan la hoja t, se escogería aquel que maximiza la reducción en impureza:

$$h = \arg\max_h \left(I_G(t) - I_G(t_h)\right) \tag{27}$$

Existen otros criterios para decidir como particionar una hoja. El lector interesado puede referirse al trabajo de Rokach y Maimon (Rokach & Maimon, 2005).

Lectura complementaria

L. Rokach, O. Maimon. (2005). "Top-down induction of decisiontrees classifiers: A survey". IEEE Transactions onSystems, Man and Cybernetics Part C: Applications and Reviews, Vol. 35(4) pp. 476-487

Selección del hiperplano

La selección del hiperplano o corte se realiza por fuerza bruta para cada una de las variables, atendiendo a su tipo. Si se trata de una variable numérica u ordinal, lo que se hace es ordenar el conjunto de datos de acuerdo a dicha variable a_j y recorrer todos los posibles hiperplanos h, seleccionando aquel corte $a_j \leq h$, es decir, variando los valores de h entre el mínimo y el máximo encontrados al ordenar todos los valores posibles. Este proceso es costoso para conjuntos de entrenamiento grandes pero conforme se va construyendo el árbol, el número de hiperplanos a comprobar se va reduciendo considerablemente.

En el caso de una variable categórica, donde no existe un orden implícito entre los posibles valores que puede tomar, es necesario generar todos los subconjuntos posibles, de forma que el hiperplano es en realidad una comparación de si un valor pertenece a un subconjunto o no. El número de subconjuntos posibles (sin incluir el subconjunto vacio o el subconjunto trivial conteniendo todos los elementos) para una variable categórica que toma c valores diferentes es $2^c - 2$, y el número de combinaciones que es necesario probar es de $2^{c-1} - 1$ (la mitad), número que crece exponencialmente con c.

4.2. Poda del árbol

Uno de los principales problemas de los árboles de decisión es que están orientados a obtener una partición del conjunto de entrenamiento "completa", es decir, intentan clasificar correctamente todos los elementos del conjunto de entrenamiento, aún cuando ello implique hacer crecer el árbol hasta profundidades muy elevadas, creando

árboles enormes con cientos o miles (o incluso millones) de nodos. El problema es que dicho árbol es muy específico para el conjunto de entrenamiento, y seguramente cometerá muchos errores para datos nuevos, causando sobreentrenamiento. Es como si se diseñara un mueble para almacenar una colección muy específica de objetos; si se presenta un objeto nuevo nunca visto anteriormente, posiblemente no encajará bien en ninguna parte, ya que el mueble es demasiado específico para la colección para la cual fue construido.

Para resolver este problema existe una segunda fase en el proceso de creación de un árbol de decisión llamada poda, dado que consiste en eliminar aquellas hojas que son las que causan el sobreentrenamiento. Para ello se calcula cual es la partición (de la cual cuelgan las hojas a eliminar o, en general, un subárbol) que aporta menor ratio entre el incremento de profundidad media del árbol y el decremento del error global de clasificación. Es decir, se eliminan aquellas hojas (o subárboles) que menos ayudan a mejorar el árbol durante el proceso de creación.

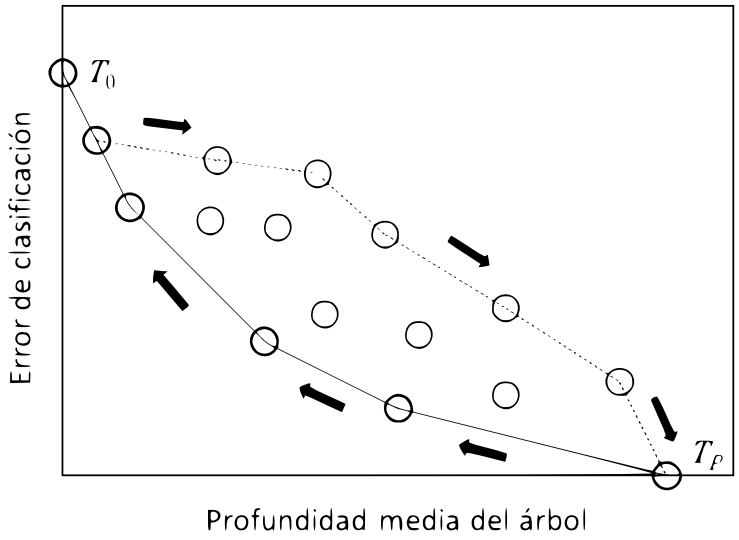
Desafortunadamente, esto no puede ser realizado durante el propio proceso de creación, ya que exigiría comprobar todos los subárboles existentes, número que crece exponencialmente a cada paso del algoritmo de crecimiento. Sin embargo, una vez se ha alcanzado el árbol puro que clasifica perfectamente el conjunto de entrenamiento, sí que es posible encontrar de forma eficiente la secuencia de sub-árboles óptimos, que son aquellos que proporcionan el menor error de clasificación en el conjunto de entrenamiento para cada una de las posibles profundidades medias.

Sea n_t el número de elementos de una hoja t y n el número total de elementos del conjunto de datos de entrenamiento. Sea $l(t)$ la profundidad de una hoja t. Entonces, definimos la profundidad media $l(T)$ de un árbol T como:

$$l(T) = \frac{1}{n} \sum_{t \in T} n_t l(t) \tag{28}$$

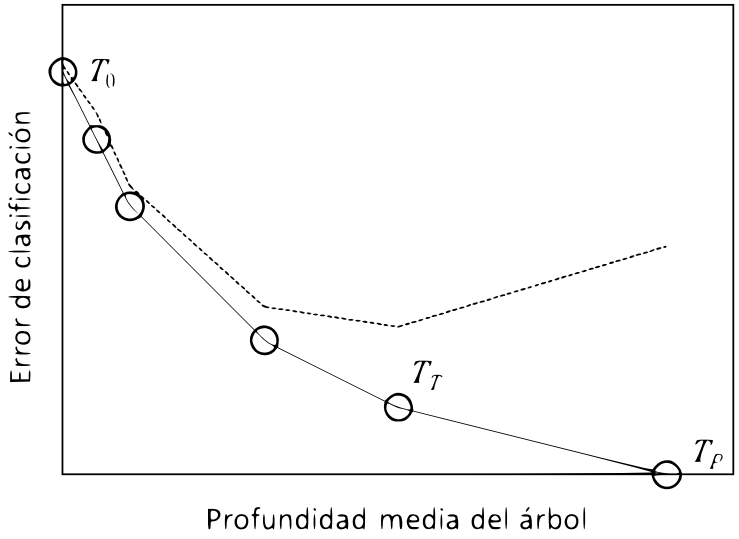
La Figura 13 muestra la idea detrás del algoritmo de poda. Cada nodo o circunferencia representa un posible árbol, siendo T_0 el árbol inicial constituido por una sola hoja que contiene todos los elementos del conjunto de entrenamiento. Durante el proceso de creación del árbol, indicado por las flechas grises, se toman decisiones que generan una secuencia de árboles, unidos por la línea de puntos, los cuales aumentan la profundidad media y van reduciendo el error de clasificación. Si otras hojas hubieran sido seleccionadas para ser particionadas durante dicho proceso, se hubiera generado otra secuencia de árboles posibles, representados por los nodos en gris. Cuando se alcanza el árbol completo T_P, el algoritmo de poda selecciona, indicado por las flechas negras, aquellos sub-árboles que son óptimos, es decir, aquellos que minimizan el error de clasificación para una profundidad media dada.

Figura 13. Procesos de creación y poda de un árbol



Es entonces cuando entra en juego el conjunto de datos de test no usado para crear el árbol de decisión. Para cada uno de los sub-árboles óptimos encontrados por el proceso de poda, se mide el error de clasificación cometido en el conjunto de datos de test. Empezando por T_0, lo más habitual es que conforme crece $l(T)$ para cada uno de los sub-árboles óptimos encontrados por el algoritmo de poda, el error cometido en el conjunto de test (indicado por la línea de puntos) también vaya descendiendo, como sucede con el conjunto de entrenamiento. Sin embargo, es posible que, a partir de cierto punto, el error en el conjunto de test se estabilice o incluso repunte, indicando que árboles más grandes son demasiado específicos y sufren de sobreentrenamiento. Así, tal como muestra la Figura 14, el resultado del proceso de creación de un árbol de decisión sería aquel sub-árbol T_T que minimiza el error cometido en el conjunto de test.

Figura 14. Selección del sub-arbol óptimo



Se recuerda al lector que en el material adicional que acompaña este libro puede encontrar ejemplos de creación y poda de árboles de decisión usando jupyter y R.

4.3. Resumen

Existen muchas razones por las cuales los árboles de decisión son uno de los modelos más utilizados en la práctica:

- Su construcción es sencilla, aunque puede ser costosa computacionalmente para conjuntos de entrenamiento muy grandes.

- El resultado obtenido es directamente interpretable, siendo también posible evaluar la importancia de cada variable utilizada en la construcción del modelo.

- Pueden combinar variables numéricas y categóricas en el mismo modelo, siendo invariantes a traslaciones y escalados de los datos, por lo que no es necesario normalizar los datos (aún cuando ésto sea recomendable en general).

- Pueden trabajar con valores perdidos, utilizando condiciones alternativas (conocidas como *surrogate splits*).

- Su implementación práctica se reduce a una serie de reglas que pueden ser fácilmente escritas como un conjunto the sentencias "if-then-else".

Por otra parte, los árboles de decisión presentan una serie de problemas que, aún siendo conocidos, no son sencillos de resolver sin una inspeccción del árbol construido (por otra parte, siempre necesaria):

- Fragmentación: a veces una hoja es particionada en dos de forma muy desequilibrada, generando dos nuevas hojas, una con pocos elementos y la otra con la mayoría. Este proceso puede generar ramas muy largas, incrementando la profundidad media del árbol y fragmentando el espacio de datos de entrada en regiones muy pequeñas, seguramente no representativas. La única solución consiste en imponer unos requisitos mínimos a la hora de particionar una hoja, de forma que las dos nuevas hojas estén equilibradas.

- Repetición: este fenómeno aparece cuando la secuencia de condiciones tomada en cada nodo interno hasta llegar a una hoja repite la misma variable, cambiando el valor por el cual se toma la decisión. Esto indica que dicha variable tiene una relación claramente no lineal con respecto a la variable objetivo, por lo que se puede intentar realizar una transformación no lineal de dicha variable a partir de un análisis de su distribución.

- Replicación: en muchos casos, los datos de entrada pueden presentar una estructura interna que es imposible de capturar si se usan hiperplanos o condiciones ortogonales, es decir, que sólo utilizan una variable en cada decisión o nodo interno. Lo que suele ocurrir es que los sub-árboles generados reproducen la estructura de variables, indicando dicho problema. La única solución posible es introducir el uso de hiperplanos oblicuos o bien realizar algún procedimiento de extracción

de características en los datos originales de forma que se reduzca la colinearidad entre variables y se elimine dicha estructura interna, capturándola en una nueva variable.

Los árboles de decisión son muy sensibles al conjunto de datos de entrenamiento, por lo que la presencia de *outliers* puede generar ramas profundas que no generalicen bien para nuevos datos. Aunque el proceso de poda seguramente eliminará dichas ramas, es mejor construir el árbol de decisión una vez se han detectado y eliminado los *outliers* del conjunto de entrenamiento.

Resumen

XXX

Glosario

Aprendizaje automático El aprendizaje automático (más conocido por su denominación en inglés, *machine learning*) es el conjunto de técnicas, métodos y algoritmos que permiten a una máquina aprender de manera automática en base a experiencias pasadas.

Aprendizaje no supervisado El aprendizaje no supervisado se basa en el descubrimiento de patrones, características y correlaciones en los datos de entrada, sin intervención externa.

Aprendizaje supervisado El aprendizaje supervisado se basa en el uso de un componente externo, llamado supervisor, que compara los datos obtenidos por el modelo con los datos esperados por éste, y proporciona retroalimentación al modelo para que vaya ajustándose y mejorando las predicciones.

Bibliografía

Y. Bengio, I. Goodfellow, A. Courville (2016). *Deep Learning*. Cambridge, MA: MIT Press.

S. O. Haykin (2009). *Neural Networks and Learning Machines,3rd Edition*. Pearson.

J. Gironés Roig, J. Casas Roma, J. Minguillón Alfonso, R. Caihuelas Quiles (2017). *Minería de datos: Modelos y algoritmos*. Barcelona: Editorial UOC.