Introducción al aprendizaje computacional

PID_00267996

Andrés Cencerrado Barraqué
Carles Ventura Royo

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Andrés Cencerrado Barraqué **Carles Ventura Royo**

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Carles Ventura Royo (2019)

Primera edición: septiembre 2019

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

Los métodos que hemos estudiado hasta ahora estaban orientados a resolver los problemas computacionalmente aplicando las estrategias propias de un **ente inteligente**: abstraer la información adecuada para representar un problema y protocolizar maneras de encontrar soluciones, explotar el conocimiento existente de que ya disponemos de un determinado sistema o, incluso, lidiar con la incertidumbre intrínseca de los problemas del mundo real. Sin embargo, estos métodos, tal como hasta ahora los hemos estudiado, dejan de lado una de las características definitorias de la inteligencia: **el aprendizaje**. Es decir, la capacidad de adaptarse automáticamente teniendo en cuenta las situaciones experimentadas.

En este módulo daremos un paso más allá para presentar, de manera introductoria, uno de los aspectos más decisivos para el éxito de la inteligencia artificial como área de conocimiento: **el aprendizaje computacional** (*Machine Learning*).

Son diversas las definiciones para saber en qué consiste el aprendizaje computacional, pero todas coinciden en un aspecto clave: la importancia de la experiencia pasada y cómo esta modifica el modelo interno que tiene un sistema inteligente de un aspecto o problema de la realidad.

> El aprendizaje computacional tiene como objetivo establecer una serie de métodos que mejoran el rendimiento del sistema a partir de su propia experiencia.

Esta subdisciplina se estudia a un nivel más profundo en las asignaturas *Aprendizaje computacional* e *Inteligencia artificial avanzada*. En este módulo, haremos un repaso más superficial e introductorio de sus principios, conceptos generales y métodos existentes más relevantes.

El primer apartado está destinado a exponer la clasificación del tipo de aprendizaje computacional más frecuentemente utilizada: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

Seguidamente, se profundiza en el aprendizaje supervisado por medio del estudio de métodos tan extendidos como el kNN, los árboles de decisión, las *Support Vector Machines* (SVM) y las redes neuronales.

Se presenta una visión del aprendizaje no supervisado a partir de los fundamentos de un tipo de algoritmos muy representativos de esta tipología: los algoritmos de agrupamiento (*k-means* y *fuzzy c-means*).

El estudio de los algoritmos de aprendizaje por refuerzo se lleva a cabo mediante el análisis de uno de los algoritmos más reconocidos: *Q-learning*. Se incluyen en esta categoría los algoritmos genéticos, los cuales se suelen utilizar también en problemas de optimización.

Finalmente, se exponen los fundamentos teóricos de la partición de datos y los protocolos de validación en el aprendizaje computacional.

Objetivos

Este módulo didáctico pretende lograr los objetivos siguientes:

1. Obtener una visión general de lo que son los sistemas de aprendizaje computacional.

2. Conocer los diferentes tipos de aprendizaje en función de la supervisión.

3. Estudiar los fundamentos teóricos de los métodos representativos del aprendizaje computacional.

4. Adquirir una base de conocimiento acerca de cómo se tratan los conjuntos de datos con objeto de validar el rendimiento de los sistemas respecto del aprendizaje realizado.

1. Taxonomía de las técnicas de aprendizaje computacional

Uno de los aspectos más importantes a la hora de considerar los métodos de aprendizaje computacional es la supervisión de la bondad de sus resultados. Es decir, de qué tipo de información se dispone acerca del resultado que tiene que dar un sistema.

Como veremos a lo largo de este módulo, unas veces dispondremos de criterios muy definidos y muestras reales que nos permitirán evaluar el comportamiento de nuestros algoritmos, otras veces tendremos aproximaciones de cariz heurístico y en otros casos no tendremos información sobre qué esperamos de la salida del sistema, sino que más bien esperamos que el sistema extraiga las características principales de las entradas para ayudarnos a entender aspectos como, por ejemplo, su estructura o las interrelaciones entre sus elementos.

Teniendo esto en cuenta, la clasificación más ampliamente aceptada de las técnicas de aprendizaje computacional es la siguiente: métodos con aprendizaje supervisado, aprendizaje por refuerzo y aprendizaje no supervisado. A continuación, repasamos los fundamentos de cada uno de estos tipos.

1.1. Aprendizaje supervisado

El aprendizaje supervisado corresponde a la situación en la que disponemos de un conocimiento completo de cuál es la respuesta que se tiene que dar en una determinada situación.

En este tipo de aprendizaje, el objetivo es conseguir un sistema basado en el conocimiento que reproduzca la salida cuando estamos en una situación que corresponda a la entrada.

En muchos casos, el problema se puede describir formalmente como un problema de optimización o como la aproximación de una función a partir de unos ejemplos (pares entrada/salida). También se suele hablar de aprendizaje a partir de ejemplos y el conjunto de ejemplos se denomina *conjunto de entrenamiento*.

1.2. Aprendizaje por refuerzo

En este caso, el conocimiento de la calidad del sistema solo es parcial. No se dispone del valor que corresponde a la salida para un determinado conjunto de valores de entrada, sino únicamente una gratificación o una penalización según el resultado que haya dado el sistema.

Por ejemplo, si un robot planifica una trayectoria y colisiona con un objeto, recibirá una penalización, pero en ningún momento hay un experto que le ofrezca una trayectoria alternativa correcta. Del mismo modo, si la trayectoria planificada por el robot no provoca ninguna colisión, recibe una gratificación. Las planificaciones que el robot haga a partir de este momento tendrán en cuenta las penalizaciones/gratificaciones recibidas.

1.3. Aprendizaje no supervisado

En el caso del aprendizaje no supervisado, no hay supervisión de casos. Solo se dispone de información de las entradas y no de las salidas. El aprendizaje tiene que extraer un conocimiento útil a partir de la información disponible.

Este tipo de aprendizaje se basa en los algoritmos de **categorización**, que permiten ordenar la información disponible, y de **agrupamiento**, que estructuran los datos de entrada en varios grupos de acuerdo con las características de cada componente, generalmente representadas en un espacio *n*-dimensional.

A continuación expondremos algunos de los algoritmos más conocidos y utilizados de cada uno de estos tipos.

2. Métodos de aprendizaje supervisado

Para formalizar este tipo de aprendizaje, consideramos un conjunto de entrenamiento C formato por N ejemplos, donde cada ejemplo es un par (x, y) en el que x es un vector de dimensión M e y es el resultado de aplicar una función f (que no conocemos) al vector x. Por tanto, tenemos N ejemplos en un espacio de dimensión M. Es decir, $X = \{x_1, \ldots, x_i, \ldots, x_N\}$. En caso de que haya un error en las medidas, tenemos que y es $f(x)$ más un cierto error ϵ. Esto es, $y = f(x). + \varepsilon$ A partir de esta información, se quiere construir un modelo que denotamos por M_C (utilizamos el subíndice porque el modelo depende del conjunto de ejemplos C). El objetivo es conseguir que el modelo M_C aplicado a un elemento x dé un resultado parecido a $f(x)$. Es decir, $M_C(x)$ es una aproximación de $f(x)$.

El aprendizaje supervisado es una técnica empleada en multitud de tipologías de problemas, de entre los que destacan los problemas de regresión, de clasificación o, incluso, los problemas de búsqueda como los que hemos estudiado anteriormente.

En los próximos subapartados, nos apoyaremos en una muestra de datos de dos conjuntos (*datasets*) que podemos encontrar en el repositorio UCI (Frank y Asuncion, 2010): el problema «iris» y el problema «mushroom», que contienen datos de características de las flores en función de las medidas del sépalo y los pétalos, y de características de las setas en función de su sombrero y tronco, respectivamente.

2.1. k Nearest Neighbors (kNN)

El **algoritmo kNN** (k vecinos más cercanos) es uno de los ejemplos clásicos del algoritmo de aprendizaje supervisado. Se utiliza principalmente para llevar a cabo clasificaciones y estas se hacen a partir de medidas de similitud o distancia. Se comparan nuevos ejemplos con conjuntos (uno por clase) de prototipos, asignando la clase del prototipo más cercano o buscando en una base de ejemplos cuál es el más cercano.

Supongamos que queremos hacer una aplicación para un almacén de flores, al que llegan diariamente miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas de las flores y nos piden que dicho sistema las clasifique automáticamente para ponerlas mediante un robot en las diferentes estanterías del almacén. Las medidas que envía el sistema láser son la longitud y el ancho del sépalo y del pétalo de cada flor. La tabla 1 muestra ejemplos de este tipo de datos.

Tabla 1. Conjunto de entrenamiento

class	*sepal-length*	*sepal-width*	*petal-length*	*petal-width*
setosa	5,1	3,5	1,4	0,2

Fuente: problema «iris» del repositorio UCI (Frank y Asuncion, 2010).

class	*sepal-length*	*sepal-width*	*petal-length*	*petal-width*
setosa	4,9	3,0	1,4	0,2
versicolor	6,1	2,9	4,7	1,4
versicolor	5,6	2,9	3,6	1,3
virginica	7,6	3,0	6,6	2,1
virginica	4,9	2,5	4,5	1,7

Fuente: problema «iris» del repositorio UCI (Frank y Asuncion, 2010).

En este algoritmo, la clasificación de los nuevos ejemplos se hace buscando el conjunto de los *k* ejemplos más cercanos entre un conjunto de ejemplos etiquetados previamente guardados y seleccionando la clase más frecuente entre las etiquetas. La generalización se pospone hasta el momento de la clasificación de los nuevos ejemplos.

Este algoritmo, en su forma más simple, deja en la memoria todos los ejemplos durante el proceso de entrenamiento y la clasificación de los nuevos ejemplos se basa en las clases de los *k* ejemplos más cercanos. Para obtener el conjunto de los *k* vecinos más cercanos, se calcula la distancia entre el ejemplo que se quiere clasificar $x = (x_1, \ldots, x_m)$ y todos los ejemplos guardados $x_i = (x_{i_1}, \ldots, x_{i_m})$. Una de las distancias más utilizadas es la euclidiana:

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

La **distancia euclidiana o euclídea** es una métrica útil en numerosas aplicaciones, especialmente si las magnitudes son lineales y la escala es uniforme. Además, es sencilla y rápida de calcular. Sin embargo, solo es aplicable a atributos numéricos. Para atributos nominales o binarios, se acostumbra a utilizar la distancia de Hamming, entre otros métodos.

Distancia de Hamming

La distancia de Hamming es:

$$dh(x, x_i) = \sqrt{\sum_{j=1}^{m} \delta(x_j - x_{ij})^2}$$

donde $\delta(x_j - x_{ij})$ es la distancia entre dos valores que corresponde a 0 si $x_j = x_{ij}$ y a 1 si son diferentes.

Ejemplo de aplicación

La tabla anterior muestra un conjunto de entrenamiento en el que tenemos que clasificar flores a partir de sus propiedades. En este ejemplo, aplicaremos el kNN para valores de *k* de 1 y 3, utilizando como medida de distancia la euclidiana. El proceso de entrenamiento consiste en guardar los datos. No tenemos que hacer nada. A partir de ahí, cuando nos llega un ejemplo nuevo como el que muestra la tabla 2, tenemos que calcular las distancias entre el nuevo ejemplo y todos los del conjunto de entrenamiento. La tabla 3 muestra estas distancias.

Tabla 2. Ejemplo de test

class	*sepal-length*	*sepal-width*	*petal-length*	*petal-width*
setosa	4,9	3,1	1,5	0,1

Fuente: problema «iris» del repositorio UCI (Frank y Asuncion, 2010).

Tabla 3. Distancias

0,5	0,2	3,7	2,5	6,1	3,5

Para el 1NN escogemos la clase del ejemplo de entrenamiento más cercano, que coincide con el segundo ejemplo (distancia: 0,2), cuya clase es la *setosa*. Para el 3NN escogemos los tres ejemplos más cercanos: primero, segundo y cuarto, con sus distancias respectivas:

0,5, 0,2 y 2,5. Sus clases corresponden a *setosa*, *setosa* y *versicolor*. En este caso, también lo asignaremos a *setosa* porque es la clase más frecuente. En los dos casos el resultado es correcto.

2.2. Árboles de decisión

Así como el algoritmo kNN es uno de los ejemplos más representativos del método basado en distancias, los árboles de decisión lo son de los métodos basados en reglas. Estos métodos adquieren reglas de selección asociadas a cada una de las clases. Dado un ejemplo de test, el sistema selecciona la clase que verifica algunas de las reglas que determinan una de las clases.

Un árbol de decisión es una manera de representar reglas de clasificación inherentes a los datos, con una estructura en árbol *n*-ario que divide los datos de manera recursiva. Cada rama de un árbol de decisión representa una regla que decide entre una conjunción de valores de un atributo básico (nodos internos) o hace una predicción de la clase (nodos terminales).

El algoritmo de los árboles de decisión básico está pensado para trabajar con atributos nominales. El conjunto de entrenamiento queda definido por $S = \{(x_1, y_1), \ldots, x(x_n, y_n)\}$, donde cada componente x_i corresponde a $x_i = (x_{i_1}, \ldots, x_{i_m})$ donde m corresponde al número de atributos de los ejemplos de entrenamiento; y el conjunto de atributos para $A = \{a_1, \ldots, a_m\}$ en el que dom(a_j) corresponde al conjunto de todos los posibles valores del atributo a_j, y para cualquier valor de un ejemplo de entrenamiento $x_{ij} \in \mathrm{dom}(a_j)$.

El proceso de construcción del árbol es un proceso iterativo, en el que en cada iteración se selecciona el atributo que hace la mejor partición del conjunto de entrenamiento. Para hacer este proceso, tenemos que comprobar la bondad de las particiones que genera cada uno de los atributos y, en un segundo paso, seleccionar el mejor. La partición del atributo a_j genera $|\mathrm{dom}(a_j)|$ conjuntos, que corresponde al número de elementos del conjunto. Existen varias medidas para comprobar la bondad de la partición. Una básica consiste en asignar la clase mayoritaria a cada conjunto de la partición, contar cuántos quedan bien clasificados y dividirlo por el número de ejemplos. Una vez calculadas las bondades de todos los atributos, escogemos el mejor.

Cada conjunto de la mejor partición pasará a ser un nuevo nodo del árbol. A este nodo se llegará por medio de una regla de tipo *atributo = valor*. Si todos los ejemplos del conjunto han quedado bien clasificados, lo convertimos en un nodo terminal con la clase de los ejemplos. En caso contrario, lo convertimos en un nodo interno y aplicamos una nueva iteración al conjunto («re-

ducido») eliminando el atributo que ha generado la partición. En caso de que no queden atributos, lo convertiremos en un nodo terminal asignándole la clase mayoritaria.

Para hacer el test, exploramos el árbol en función de los valores de los atributos del ejemplo de test y las reglas del árbol hasta llegar al nodo terminal, y damos como predicción la clase del nodo terminal al que llegamos.

Ejemplo de aplicación

Supongamos que queremos hacer una aplicación para teléfonos móviles que ayude a los aficionados a coger setas a discernir las venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y color del tronco. La tabla 4 muestra un ejemplo de un conjunto de estos datos, que usaremos como conjunto de entrenamiento.

Tabla 4. Conjunto de entrenamiento

class	***cap-shape***	***cap-color***	***gill-color***
poisonous	*convex*	*brown*	*black*
edible	*convex*	*yellow*	*black*
edible	*bell*	*white*	*brown*
poisonous	*convex*	*white*	*brown*
edible	*convex*	*yellow*	*brown*
edible	*bell*	*white*	*brown*
poisonous	*convex*	*white*	*pink*

Fuente: problema «mushroom» del repositorio UCI (Frank y Asuncion, 2010).

Para construir un árbol de decisión a partir de este conjunto, tenemos que calcular la bondad de las particiones de los tres atributos: *cap-shape*, *cap-color* y *gill-color*. El atributo *cap-shape* nos genera una partición con dos conjuntos: uno para el valor *convex* y otro para *bell*. La clase mayoritaria para el conjunto de *convex* es *poisonous* (tres elementos son *poisonous* y dos elementos son *edible*) y la de *bell* es *edible* (los dos elementos son *edible*). Su bondad es bondad(*cap-shape*) = (3 + 2)/7 = 0,71. Si hacemos el mismo proceso para el resto de atributos obtenemos: bondad(*cap-color*) = (1 + 2 + 2)/7 = 0,71 y bondad(*gill-color*) = (1 + 3 + 1)/7 = 0,71.

El paso siguiente consiste en seleccionar el mejor atributo. Hemos obtenido un empate entre los tres atributos y podemos escoger cualquiera. Escogemos *cap-color*. Los nodos generados por el conjunto de *brown* y *yellow* son terminales y los asignamos a las clases *poisonous* y *edible*, respectivamente. Esto es así porque los dos conjuntos obtienen una bondad de 1. El nodo *white* nos queda con los datos que muestra la tabla 5. Este conjunto lo obtenemos de eliminar el atributo *cap-color* de los ejemplos que tienen el valor *white* para el atributo *cap-color*. Volvemos a iterar. La bondad de las nuevas particiones es: bondad(*cap-shape*) = (2 + 2)/4 = 1 y bondad(*gill-color*) = (2 + 1)/4 = 0,75.

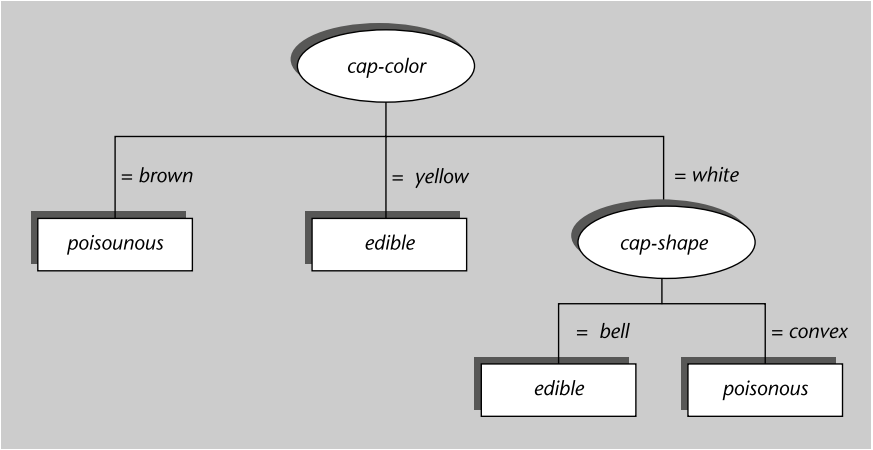
El mejor atributo es *cap-shape*, que genera dos nodos terminales con las clases *edible* para *bell* y *poisonous* para *convex*. La figura 1 muestra el árbol construido en este proceso.

Tabla 5. Conjunto de la segunda iteración

class	***cap-shape***	***gill-color***
edible	*bell*	*brown*
poisonous	*convex*	*brown*
edible	*bell*	*brown*

class	cap-shape	gill-color
poisonous	convex	pink

Figura 1. Árbol de decisión



Para etiquetar un ejemplo de test como el que muestra la tabla 6 tenemos que recurrir al árbol partiendo de la raíz y escogiendo las ramas correspondientes a los valores de los atributos de los ejemplos de test. Para este caso, miramos el valor del atributo *cap-color* y bajamos por la rama que corresponde al valor *brown*. Llegamos a un nodo terminal con clase *poisonous*, con lo que lo asignaremos al ejemplo de test como predicción. Esta predicción es correcta.

Tabla 6. Ejemplo de test

class	cap-shape	gill-color	gill-color
poisonous	convex	brown	black

Fuente: problema «mushroom» del repositorio UCI (Frank y Asuncion, 2010).

Este algoritmo tiene la gran ventaja de la facilidad de interpretación del modelo de aprendizaje. Un árbol de decisión nos da una información clara de la toma de decisiones y de la importancia de los diferentes atributos involucrados. Por esta razón se ha utilizado mucho en varios campos, como, por ejemplo, el financiero.

2.3. Support Vector Machines (SVM)

Uno de los métodos de clasificación más utilizados clásicamente son las máquinas de vectores de soporte[1], basadas en la estadística. Este método permite clasificar objetos descritos mediante datos numéricos que consideran solo la existencia de dos clases (como podría ser el caso del problema «mushroom» con las clases *poisonous* y *edible*).

(1)En inglés, *Support Vector Machines* (SVM).

Ved también

En este apartado haremos una breve introducción a las SVM lineales y no lineales, sin entrar en detalles matemáticos de la formulación, la cual se trata en las asignaturas *Aprendizaje computacional* e *Inteligencia artificial avanzada*.

2.3.1. SVM lineales

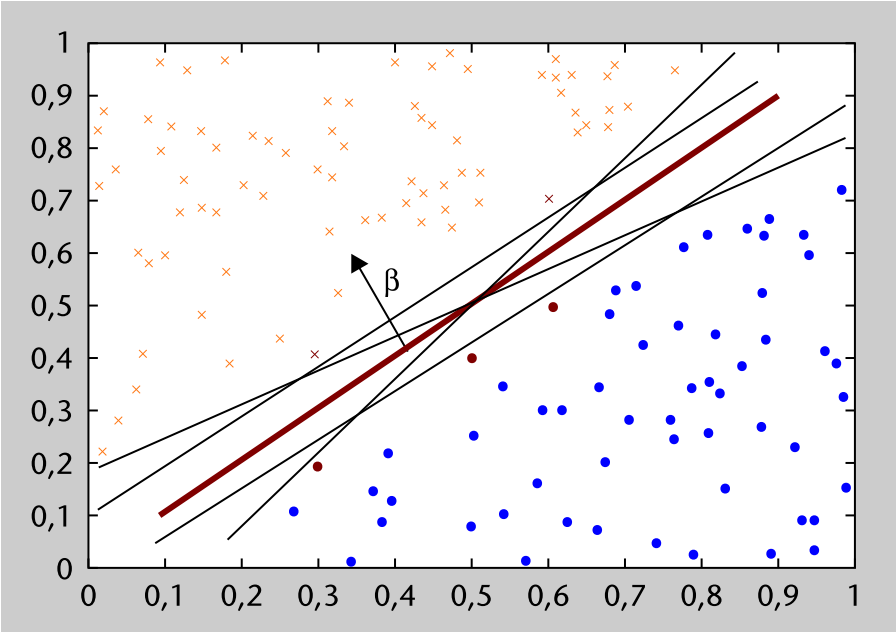
Las máquinas lineales de vectores de soporte se basan en la construcción de un hiperplano en el espacio de los datos que separa los objetos que pertenecen a una clase de los que pertenecen a la otra. Podemos encontrarnos con dos casos:

1) Los objetos son linealmente separables: es posible encontrar un hiperplano que separe completamente los objetos de las dos clases.

2) Los objetos no son linealmente separables: no es posible encontrar un hiperplano que separe completamente los objetos de las dos clases, por lo que deberemos buscar un hiperplano que separe a la mayoría de los objetos.

Cuando las clases son linealmente separables, la mejor regla de clasificación corresponderá a encontrar el hiperplano que defina un espacio más ancho con los objetos. En la figura 2, se puede ver un ejemplo de clases linealmente separables, donde se muestra un conjunto de puntos que pertenecen a dos clases (cruces y círculos). El problema se reduce a encontrar el hiperplano separador más adecuado. Observad que hay infinitos hiperplanos que separan perfectamente estas dos clases. En la figura, se han dibujado unos cuantos (hiperplanos negros), pero podemos intuir que el hiperplano que más interesa es aquel que deja más distancia entre cada clase y el hiperplano separador, es decir, en la figura 2, el hiperplano óptimo que separa los objetos de las dos clases es el que está representado en granate.

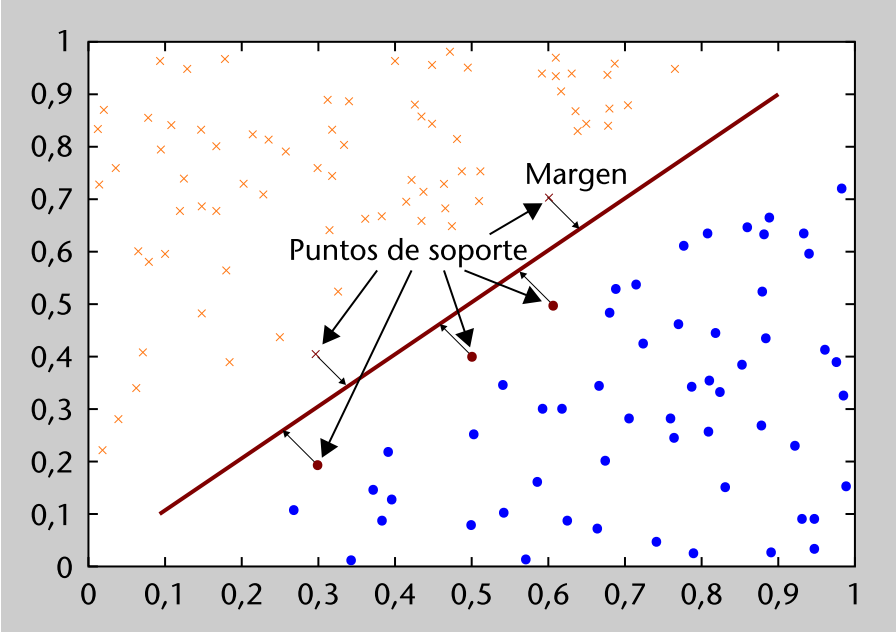
Figura 2. Ejemplo de clases linealmente separables



Con esta idea intuitiva, definiremos el concepto de **margen** como la distancia entre el punto más cercano de cada clase y el hiperplano separador. Y denominaremos **vectores de soporte** (*support vectors*) a los objetos que están a esa distancia del hiperplano.

La figura 3 muestra gráficamente el concepto de vectores de soporte y de margen. Observad que los parámetros que definen el hiperplano dependerán exclusivamente de estos vectores de soporte, que son los que realmente determinan la frontera de separación.

Figura 3. Vectores de soporte y margen



Fijaos que a pesar de que el ejemplo dado es en un espacio de dos dimensiones, el mismo concepto es extensible a cualquier espacio de dimensiones.

Cuando las clases no son linealmente separables, hay que tener en cuenta el solapamiento entre clases. La idea es la misma, situar el hiperplano de forma que la distancia a los objetos cercanos sea lo mayor posible. Sin embargo, ahora sí puede haber algunos puntos en el lado incorrecto del hiperplano.

2.3.2. SVM no lineales

Cuando las clases no son linealmente separables, también hay otra alternativa: la utilización de **máquinas no lineales de vectores de soporte**. Estas máquinas consideran la transformación del espacio de los datos originales en un nuevo espacio diferente y más grande, donde la separación lineal de los elementos sea posible o, al menos, se reduzca el número de objetos que estaban en el lado incorrecto del hiperplano del espacio original.

Así pues, se considera una función Φ que transforma los atributos de los objetos del espacio original a un nuevo espacio donde se aplica una SVM lineal (tal como se ha descrito anteriormente).

La formulación matemática de las SVM permite que no haga falta la definición de la función Φ que transforma los atributos, sino que basta con definir el producto escalar de parejas de elementos transformados. La función que define este producto es lo que se denomina *función núcleo* (kernel).

En la figura 4, podemos ver un ejemplo de clases no linealmente separables, puesto que no es posible encontrar en el espacio original un hiperplano que separe completamente ambas clases. Sin embargo, se puede observar que los datos siguen una distribución circular. En este caso, si utilizamos la función núcleo (kernel) gaussiano, entonces los objetos del nuevo espacio transformado son linealmente separables.

Kernels no lineales

Los kernels no lineales más habituales son el polinómico y el gaussiano.

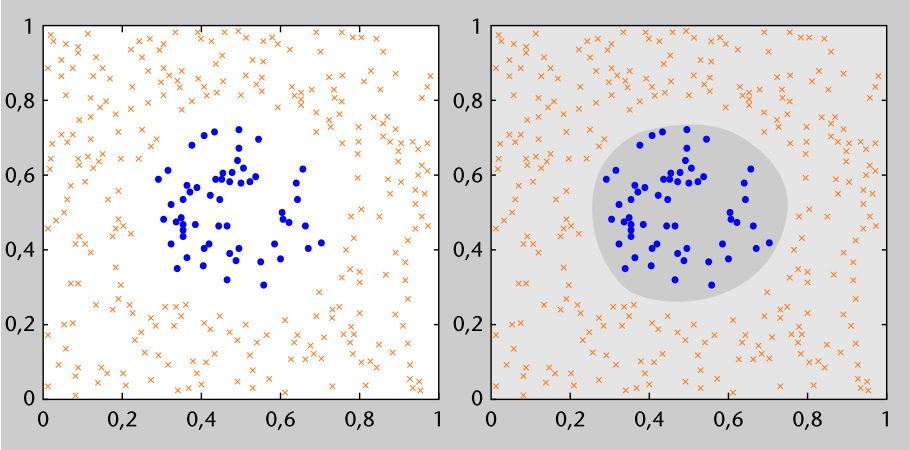
El **kernel polinómico** se define como:

$$\kappa(x_i, x_j) = p(\kappa_1(x_i, x_j))$$

El **kernel gaussiano** se define como:

$$\kappa(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

Figura 4. Ejemplo de clases no linealmente separables



Hay dos kernels no lineales de aplicación general que aparecen en todas las implementaciones de los métodos basados en kernels: el polinómico y el gaussiano.

2.4. Redes neuronales

El objetivo de las redes neuronales es crear un análogo computacional a las neuronas biológicas para intentar crear inteligencia en un ordenador. En este sentido, se definen unos elementos denominados *neuronas*, o simplemente unidades con conexiones a otras unidades mediante las cuales se propagan señales.

En este apartado, haremos una introducción de las bases teóricas de este método y cómo se relaciona con la disciplina del *Deep Learning*, muy reconocida y ampliamente aplicada durante la última década.

Ved también

En la asignatura *Inteligencia artificial avanzada* se estudia en profundidad esta temática, analizando varias arquitecturas y exponiendo casos de aplicación real, como, por ejemplo, los clasificadores de imágenes.

Una red neuronal básica se compone de los elementos siguientes:

- Una capa de unidades de entrada, que reciben las variables disponibles del exterior para tratar el problema: los píxeles de una imagen, la posición de un objeto, los valores de ciertos productos, etc.

- Una capa de unidades de salida, compuesta por una o más unidades que producen una salida al exterior, que es el «resultado» de la red: la clase de imagen, el valor de la tensión eléctrica para accionar un motor, si se tiene que comprar o vender un producto, etc.

- Una capa de unidades ocultas, que reciben conexiones de la capa de entrada y se conectan a las unidades de salida.

- Finalmente, el conjunto de conexiones entre capas. En general, las conexiones entre las unidades son unidireccionales.

Como podéis ver, las unidades se organizan en capas. En el tipo de red neuronal más sencilla, la red neuronal prealimentada o perceptrón, las conexiones siempre van de las unidades de la capa de entrada a las de la capa oculta, y de ahí a la capa de salida. Las conexiones no van hacia atrás.

El funcionamiento del perceptrón consiste básicamente en la **propagación** hacia adelante (*forward*) de las señales de entrada, lógicamente modificadas a medida que atraviesan las capas.

Cuando todas las unidades de una capa están conectadas a todas las unidades de la capa siguiente se dice que esta segunda capa está **completamente conectada** (*fully connected*). En el perceptrón las capas son así, pero en otras arquitecturas no pasa lo mismo.

De manera simplificada, el funcionamiento del perceptrón es el siguiente:

1) Las unidades de entrada reciben valores del exterior y se activarán –es decir, producirán un determinado valor a la salida– o no en función de la entrada recibida.

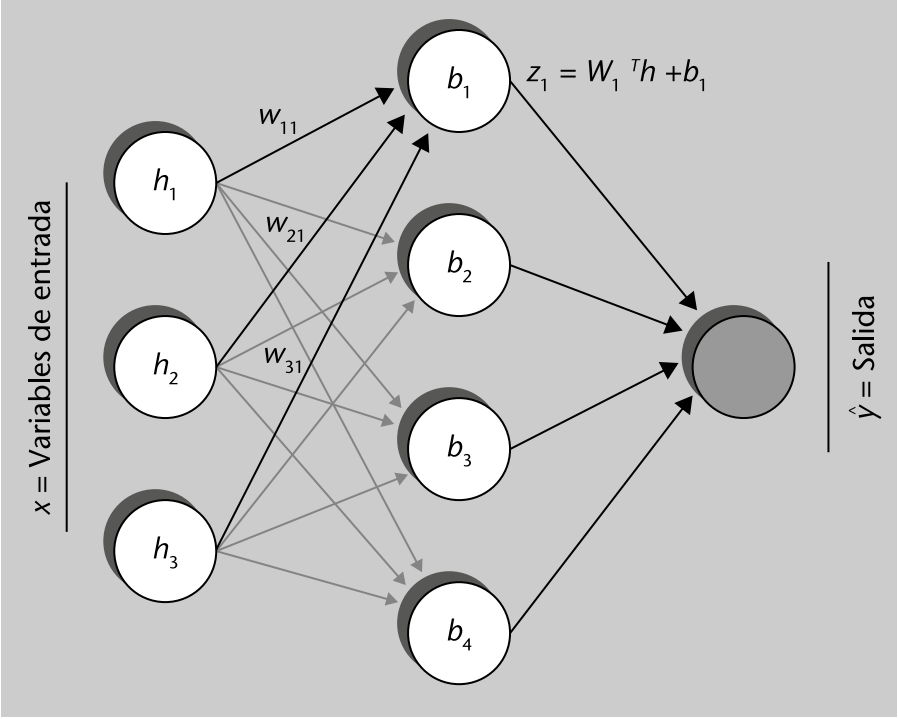
2) Las salidas de la capa de entrada son, a su vez, las entradas de la capa oculta, de forma que estas unidades reciben un conjunto de entradas frente a las cuales reaccionarán. Para hacerlo, cada unidad de la capa oculta tiene un vector de

pesos, un valor para cada conexión entrante, que combina con las señales correspondientes y, como resultado, producen la activación o no de la salida de cada unidad oculta.

3) Finalmente, las unidades de la capa de salida también reciben las señales de la capa oculta, realizan una operación con estas y sus propios vectores de pesos y, como resultado, calculan su propia salida, que será el resultado de la red.

La figura 5 muestra esquemáticamente la estructura de una red neuronal básica.

Figura 5. Red neuronal básica



La manera que tiene una unidad de combinar sus entradas X con su vector de pesos W para calcular su salida z viene dada, generalmente, por la expresión:

$$z = W^T X + b$$

donde b es un valor escalar, denominado sesgo (*bias*), necesario para garantizar un valor de base. El valor $z \in \mathbb{R}$ se utiliza, a su vez, como entrada para la función de activación, que es la que decide cuál es la salida final.

La función de activación más utilizada en las unidades de entrada y ocultas es el llamada **ReLU**[2], que es tan sencilla como:

(2)Del inglés, *Rectified Linear Unit*.

$$g(z) = \max(0,z)$$

Es decir, la salida es 0 si la entrada es negativa, y es igual a *z* si esta es positiva.

Funciones de activación

La elección de la función de activación es clave en este tipo de sistemas. Dependiendo del problema, podemos encontrar una gran diversidad: funciones de activación lineales, logística sigmoidea, etc.

2.4.1. Entrenamiento

Como se puede inferir, los valores *W* y *b* son los que controlan el comportamiento de la red. Una de las características más potentes de las redes neuronales es que son ellas mismas las que aprenden los valores óptimos para estas variables con el fin de llevar a cabo su propósito de la mejor manera posible.

Cuando se trabaja con redes neuronales, se distinguen dos fases:

1) Fase de entrenamiento: la red recibe datos de que tiene que aprender y va ajustando los pesos y sesgos.

2) Fase de ejecución: la red se utiliza para llevar a cabo la tarea tal como ha aprendido.

Como método de aprendizaje supervisado que son las redes neuronales, el entrenamiento del perceptrón requiere un conjunto de datos etiquetados. Los pesos y sesgos se inicializan a valores aleatorios pequeños (alrededor de 0,1). A continuación, se van inyectando los ejemplos de entrenamiento en la red y se analiza la diferencia entre la salida obtenida y la salida esperada (según la etiqueta asociada a cada ejemplo).

Esta diferencia entre la salida obtenida y la salida esperada se expresa mediante una función de coste *C*. En redes con una sola salida, se puede utilizar como *C* la función de error cuadrático. El objetivo del entrenamiento es reducir los valores de *C*, es decir, la diferencia entre la salida obtenida y la esperada. Por eso, hay que ajustar gradualmente los pesos y sesgos de cada capa calculando el **gradiente de la función de coste** respecto de los pesos y sesgos de la unidad de salida:

Gradiente de función

Los gradientes de una función *f* de *n* variables son las derivadas de *f* respecto de cada una de que depende. Por ejemplo, si tenemos *f(x, y)*, sus gradientes serán las derivadas de *f* respecto de *x* e *y*, y normalmente se representan con la forma:

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$$

$$\frac{\partial C}{\partial \widehat{W}}$$

donde $\widehat{W} = \{W, b\}$, es decir, se integran pesos y sesgo en un único vector para facilitar las operaciones. Así, pues, cuando se determinan los $\widehat{W}$ óptimos para reducir la función de coste *C*, se está ajustando la capa de salida. Sin embargo, las capas anteriores no han recibido ningún ajuste. Aquí entra en juego el proceso denominado **propagación hacia atrás** (*backpropagation*): una vez ajustada la capa de salida, se procede a ajustar la capa oculta mediante el mismo procedimiento, y así se van ajustando las capas desde la salida hacia la entrada, por este motivo se hace «hacia atrás».

De este modo se acaban ajustando todos los parámetros de la red, haciendo que el conjunto de parámetros produzca el menor error posible con los datos de entrenamiento. Este **error de entrenamiento** puede ser diferente de cero en función de los datos y la complejidad del modelo. Un modelo más complejo será capaz de aprender mejor los datos de entrenamiento, pero, como se verá más adelante, no es conveniente llevar esta idea al extremo para conseguir un error de entrenamiento igual a cero.

2.4.2. Aprendizaje profundo

En este apartado hemos estudiado la red neuronal sencilla, el perceptrón, en la que solo hay tres capas. Una red con una sola capa oculta como esta es capaz de aprender cualquier conjunto de datos finito, siempre que tenga un número suficientemente grande de unidades en la capa oculta.

El problema de este planteamiento, aumentar la anchura de la capa oculta, es doble:

- Es una solución computacionalmente costosa, puesto que puede requerir una capa con muchos millares o, incluso, millones de unidades.

- Con un sistema así solo se aprenden las combinaciones de variables presentes en los datos de entrenamiento. Así, pues, su poder de generalización es muy rudimentario.

Una estrategia alternativa para aumentar la complejidad de una red neuronal es incrementar el número de capas ocultas. Como resultado, se obtiene una red que es capaz de generalizar mejor los nuevos datos, puesto que en cada capa se modela un nivel de abstracción superior al de la anterior. Así, pues, al final, la red es capaz de detectar características más generales.

Esta es la idea del aprendizaje profundo: crear redes neuronales con varias capas ocultas. Por eso se denomina «profundo». Generalmente, se considera profunda una red con diez o más capas.

Hasta hace unos años (alrededor de 2010) no se habían utilizado estos sistemas de manera generalizada por la dificultad y el coste computacional de entrenarlos. Sin embargo, diferentes adelantos teóricos y técnicos han favorecido las técnicas de aprendizaje profundo y han revolucionado la IA.

3. Métodos de aprendizaje no supervisado

El aprendizaje no supervisado corresponde a una situación en la que no se dispone de información del resultado que tendría que dar el sistema para un ejemplo concreto. Los algoritmos de agrupamiento son unos claros representantes de estos métodos y por este motivo daremos dos ejemplos concretos.

3.1. Algoritmos de agrupamiento

La tarea de agrupamiento de los datos es no supervisada, puesto que los datos que se proporcionan al sistema no llevan asociada ninguna etiqueta o información añadida por un revisor humano. Por el contrario, es el mismo método de agrupamiento el que tiene que descubrir las nuevas clases o grupos a partir de los datos recibidos.

A menudo, el agrupamiento de los datos precede a la clasificación de nuevos datos en alguno de los grupos obtenidos en el agrupamiento. Por esta razón, el agrupamiento y la clasificación de datos están estrechamente relacionados.

Una de las características más importantes de los algoritmos de agrupamiento es que permiten organizar datos que, en principio, no sabe o no puede clasificar, evitando criterios subjetivos de clasificación, lo cual produce una información muy valiosa a partir de datos desorganizados.

Una característica común a casi todos los algoritmos de agrupamiento es que no son capaces de determinar por sí mismos el número de grupos idóneo, sino que hay que fijarlo por adelantado o utilizar algún criterio de cohesión para saber cuándo hay que detenerse (en el caso de los jerárquicos). En general, esto requiere probar con diferentes números de grupos hasta obtener unos resultados adecuados.

A continuación, se presentan brevemente un par de los algoritmos de agrupamiento más estudiados.

3.1.1. *k-means*

El algoritmo de **agrupamiento k-medias**[3] busca una partición de los datos tal que cada punto esté asignado al grupo con el centro (llamado *centroide*, puesto que no necesariamente tiene que ser un punto de los datos) más cercano. Se le tiene que indicar el número *k* de agrupamientos (o clústeres) que queremos, puesto que por sí mismo no es capaz de determinarlo.

[3] *k-means*, en inglés.

En esencia, el algoritmo es el siguiente:

1) Elegir *k* puntos al azar como centroides iniciales. No necesariamente tienen que pertenecer al conjunto de datos, aunque sus coordenadas tienen que estar en el mismo intervalo.

2) Asignar cada punto del conjunto de datos al centroide más cercano y formar así *k* grupos.

3) Recalcular los nuevos centroides de los *k* grupos, que estarán en el centro geométrico del conjunto de puntos del grupo.

4) Volver al paso 2 hasta que las asignaciones a grupos no varíen o se hayan superado las iteraciones previstas.

Aunque se trata de un algoritmo sencillo y rápido, dado que solo tiene en cuenta la distancia a los centroides, puede fallar en algunos casos (nubes de puntos de diferentes formas o densidades), puesto que en general tiende a crear esferas de medida similar que realizan la partición del espacio.

La figura 6 muestra un ejemplo de partición de datos producido por el algoritmo de un determinado conjunto etiquetado según cinco categorías diferentes (correspondientes a las diferentes formas de los puntos). Se puede observar que, por ejemplo, algunos puntos del grupo superior derecho (triángulos) se han asignado al superior izquierdo (círculos) para estar más cercanos al centroide, sin tener en cuenta el espacio que los separa. El mismo problema se da en otros puntos. Además, el resultado puede variar de una ejecución a otra, puesto que depende de los centroides generados aleatoriamente en el primer paso del algoritmo.

Figura 6. Partición de un conjunto de datos mediante *k-means*.

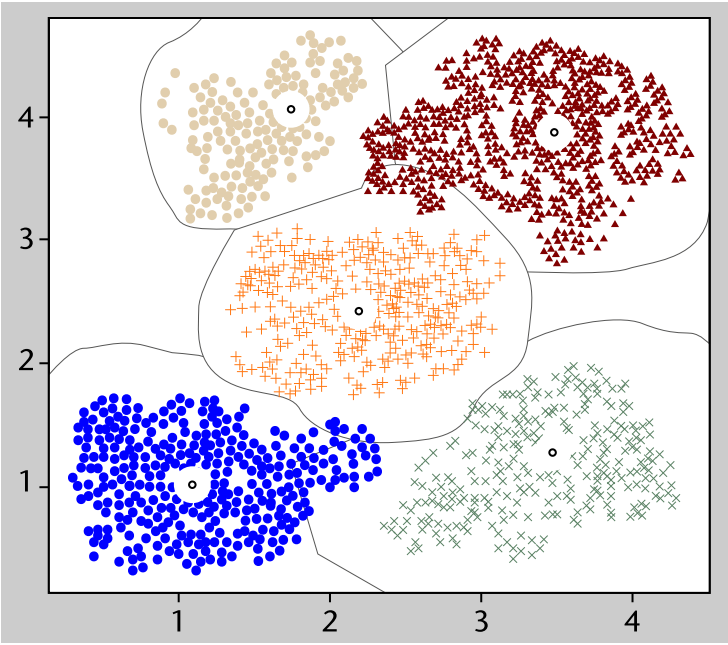


Figura 6

Los puntos rodeados de una zona blanca representan los centroides de los grupos obtenidos.

3.1.2. *Fuzzy c-means*

Resulta lógico pensar que, en *k*-medias, un punto que esté junto al centroide estará más fuertemente asociado a su grupo que un punto que esté en el límite con el grupo vecino. El **agrupamiento difuso** (*fuzzy clustering*) representa este grado diferente de vinculación, haciendo que cada dato tenga un grado de pertenencia a cada grupo, de forma que un punto junto al centroide puede tener 0,99 de pertenencia a su grupo y 0,01 al vecino, mientras que un punto junto al límite puede tener 0,55 de pertenencia a su grupo y 0,45 al vecino. Esto es extensible a más de dos grupos, entre los cuales se repartirá la pertenencia de los datos. Se puede decir que el **agrupamiento discreto** (no difuso) es un caso particular del agrupamiento difuso donde los grados de pertenencia son 1 para el grupo principal y 0 para los restantes.

El algoritmo *fuzzy c-means* (*c*-medias difuso) es prácticamente idéntico al algoritmo *k*-medias que hemos visto anteriormente. Las diferencias principales son:

- Cada dato x tiene asociado un vector de k valores reales que indican el grado de pertenencia $m_x(k)$ de este dato a cada uno de los k grupos. El grado de pertenencia de un punto a un grupo depende de su distancia al centroide correspondiente. Habitualmente, la suma de los grados de pertenencia de un dato es igual a 1.

- En lugar de crear k centroides aleatoriamente, se asigna el grado de pertenencia de cada punto a cada grupo aleatoriamente y, después, se calculan los centroides a partir de esta información.

- El cálculo de los centroides está ponderado por el grado de pertenencia de cada punto al grupo correspondiente $m_x(k)$.

A grandes rasgos, las ventajas e inconvenientes del algoritmo *c*-medias difuso son los mismos que los de *k*-medias: simplicidad, rapidez, pero no determinismo y excesiva dependencia de la distancia de los puntos a los centroides, sin tener en cuenta la densidad de puntos de cada zona (por ejemplo, espacios vacíos). Se trata de un algoritmo especialmente utilizado en el procesamiento de imágenes.

4. Métodos de aprendizaje por refuerzo

El aprendizaje por refuerzo se puede considerar como el tipo de aprendizaje del que principalmente las diferentes especies vivas han sacado provecho a lo largo de millones de años en la naturaleza. Como se ha comentado antes, este tipo de aprendizaje se caracteriza por no tener un criterio explícito y específicamente definido para evaluar si la respuesta que da un sistema es correcta o no, sino que el sistema recibe una retroalimentación desde el entorno en el que interactúa. Esta retroalimentación le permite al sistema conocer el resultado de sus interacciones y evaluarlo siguiendo unos criterios que permiten identificar el grado de éxito que ha tenido en su propósito inicial.

Esta analogía hace que muchos de los algoritmos por refuerzo existentes estén considerados dentro de la categoría de **algoritmos bioinspirados**.

Como algoritmo representativo de la técnica de aprendizaje por refuerzo, en esta sección estudiaremos en primer lugar los métodos *Q-learning* y, a continuación, uno de los algoritmos bioinspirados que ha tenido más popularidad y aplicaciones en las últimas décadas, los algoritmos genéticos.

4.1. Métodos *Q-learning*

En el aprendizaje por refuerzo, el agente o sistema tiene que aprender y actuar, guiar su aprendizaje en función de las recompensas que recibe como consecuencia de sus acciones. Las recompensas pueden ser de naturaleza diversa, aunque siempre orientadas a premiar el avance hacia el objetivo del agente: puntos conseguidos en un juego, kilómetros conducidos hacia el destino sin accidentes, piezas acopladas por un robot, etc.

De manera más formal, un sistema de aprendizaje por refuerzo se compone de los elementos siguientes:

- **Un conjunto de estados del problema, *S***. En el juego del tres en raya, por ejemplo, *S* serán todos los estados posibles de cruz, círculo o casilla vacía, un total de 3^9 estados posibles, a pesar de ser un juego muy sencillo. Sin embargo, el número de estados posibles se dispara rápidamente a unas 10^{40} posiciones válidas para el ajedrez o unas 10^{100} para el Go. Si salimos del ámbito de los juegos, el número de estados posibles es imposible de calcular y, en la práctica, infinito: ¿cuántos estados posibles hay para el problema de la conducción automática?, ¿cuántas combinaciones de vehículos, combinaciones ambientales, carretera, estado del vehículo, peatones, etc.?

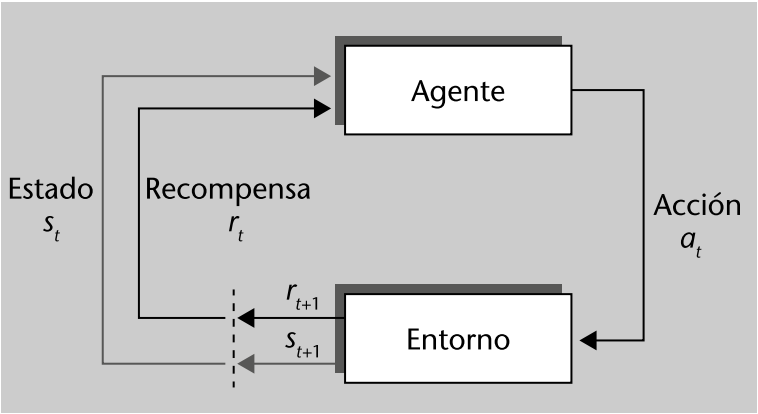
- **Un conjunto de posibles acciones, *A*.** En un juego serían los posibles movimientos. Con un robot *A* son los movimientos que puede ejecutar; en un sistema de conducción automática, las diferentes acciones que puede ordenar al coche: acelerar, frenar, girar, etc.

- **Una función de recompensa, *R***, que devuelve un número real en recompensa por la acción tomada por el agente en un estado determinado, es decir, R: $S \times A \rightarrow \Re$. Es muy importante destacar que la recompensa depende de la acción realizada y del estado actual, puesto que a veces será bueno que el coche acelere, pero otras veces no.

- En teoría, un agente de un sistema de aprendizaje por refuerzo (*reinforcement learning*, RL) debería aprenderse una **tabla de recompensas** $Q = S \times A$ para saber perfectamente cuál es la acción que se tiene que aplicar en cada estado del sistema. Sin embargo, para casi cualquier problema *S* la tabla tendrá un tamaño infinito, por lo que es imposible almacenar todo ese conocimiento en una tabla. Por este motivo, se han propuesto diferentes métodos para aprender aproximaciones razonablemente buenas a *Q*, métodos que reciben el nombre de *Q-learning*.

La figura 7 resume los elementos de un sistema de aprendizaje por refuerzo que se acaban de explicar.

Figura 7. Diagrama de control para un sistema de aprendizaje por refuerzo



A pesar de que hay varias estrategias para *Q-learning*, los sistemas que utilizan redes neuronales profundas están obteniendo grandes éxitos en diferentes ámbitos, como el sistema DQN, que aprende a jugar autónomamente a juegos de consola; el sistema AlphaGo, que venció al campeón mundial de Go, o numerosos sistemas de conducción automática y robótica, entre otros.

4.2. Algoritmos genéticos

Los algoritmos genéticos son algoritmos de búsqueda estocásticos inspirados en los fenómenos naturales de herencia genética, donde el mejor es el que sobrevive (supervivencia de las especies). En una población de individuos, las

nuevas generaciones están más adaptadas al medio que las precedentes y, de media, las nuevas poblaciones serán más rápidas, con un mayor grado de mimetismo, etc., que las anteriores porque esto es lo que les permite sobrevivir.

La analogía con el proceso biológico de la evolución dirige todos los pasos de los algoritmos genéticos. Así, consideraremos poblaciones de individuos representados por sus cromosomas (formados por genes) y, dado un cromosoma, consideraremos su entrecruzamiento con otro cromosoma o su mutación. Los entrecruzamientos y las mutaciones conducirán a nuevas generaciones de poblaciones.

Como todos los métodos de búsqueda, los algoritmos genéticos permiten encontrar una solución a un problema dado. No obstante, dado que son algoritmos estocásticos, normalmente no encuentran la mejor solución del problema, sino que hallan una que se aproxima a la solución óptima.

4.3. Aplicación e implementación

Para poder aplicar este método a un problema concreto, tenemos que empezar encontrando una manera de codificar las diversas alternativas que se han de considerar en la solución (cuáles son las posibles soluciones de un problema). Por ejemplo, si consideramos el problema del camino más corto entre dos poblaciones, las alternativas que se han de considerar son todos los caminos que conectan a estas dos poblaciones.

La codificación de las alternativas se basa en una secuencia de símbolos de longitud finita. Los símbolos que se usan para la codificación se denominan *genes* y la secuencia se llama *cromosoma*. Por su parte, el conjunto de genes se denomina *pool*. Así, tenemos que cada posible alternativa será un cromosoma formado por genes.

Además de seleccionar una representación de las alternativas, necesitamos una función que evalúe las diferentes alternativas, de forma que la mejor de ellas (la solución) tenga el mejor valor. Esta función se llama *función de evaluación* (*fitness function*). En el caso del camino más corto, podemos usar la longitud del trayecto como función (o la inversa de su longitud si queremos una función para maximizar).

A partir de estos elementos, los algoritmos genéticos buscan la solución en un proceso iterativo que consiste en crear una población de cromosomas (un conjunto de posibles soluciones), evaluar los cromosomas de la población mediante la función de evaluación y seleccionar los mejores. Estos cromosomas serán la semilla de la nueva población que se creará en el paso de iteración siguiente. El esquema general se presenta de la forma siguiente.

```
función Algoritmos genéticos retorna cromosoma se
```

```
    inicializa la población de cromosomas
    evalúa todos los cromosomas de la población
    selecciona el mejor cromosoma
    mientras no se satisface el criterio de acabamiento hacer
        crea una nueva población de cromosomas
        evalúa todos los cromosomas de la población
        selecciona el mejor cromosoma
    fmientras
    retorna el mejor cromosoma
ffunción
```

A continuación, describiremos con más detalle esta función. Consideraremos que la población contiene m individuos y que cada individuo está formado por una secuencia de longitud n.

Denotaremos el conjunto de genes por G. Así, un cromosoma c satisface $c \in G^n$. La población en la iteración k-ésima la denotamos por $p^{(k)}$ y será de la forma $p^{(k)} = \{c_1^{(k)}, \ldots, c_m^{(k)}\}$ donde $c_1^{(k)} \in G^n$ Cuando tenemos una población concreta p_{ij} será el gen j-ésimo del individuo i-ésimo.

```
función Algoritmos genéticos (f: función de evaluación) retorna cromosoma se
    k:=1;
    p(k):=inicializa();
    av:=evalúa(f,p(k));
    m(k):=selecciona_mejor(av, p(k))
    mientras no se tiene que acabar (k, m(k), m(k-1)) hacer
        k:=k+1;
        p(k):=crea_nueva_población (av, p(k-1)); av:=evalúa(f,p(k));
        m(k):=selecciona_mejor(av, p(k));
    fmientras;
    retorna m(k);
ffunción;
```

En este esquema, `av` corresponde a las evaluaciones de los cromosomas, y `m(k)` es el mejor cromosoma de la población.

Ahora describimos cada uno de los pasos que nos aparecen en este proceso.

1) Inicialización: se tiene que crear el conjunto de cromosomas inicial que corresponderá a la primera población $p^{(1)}$. Normalmente se hace de manera aleatoria. Esto es, para cada $x^{(k)}$ se define una secuencia de dimensión n con genes de G elegidos al azar.

2) Evaluación: para cada uno de los cromosomas de la población, se produce una valoración de hasta qué punto es buena la solución correspondiente. La evaluación reconstruirá la alternativa a partir del cromosoma y después le aplicará la función de evaluación. En el caso del camino más corto, a partir de una lista de bits, reconstruirá el trayecto y después se aplicará la función seleccionada para indicar hasta qué punto es bueno este camino.

3) Selección: la evaluación de los cromosomas permite elegir cuál es el mejor de todos los que hay en la población.

4) Creación: por medio de la información codificada en los cromosomas y usando la evaluación de cada uno, se crea una nueva población. El proceso de construir los nuevos cromosomas es una analogía de la evolución en el medio natural. Se utilizan operaciones de entrecruzamiento y mutación.

Seguidamente se dan los algoritmos para cada una de estas funciones.

```
función inicializa retorna población se para i:=1 hasta m hacer
    para j:=1 hasta n hacer
        pij:=elige gen aleatoriamente de (G)
    fpara;
fpara;
retorna p;
ffunción;

función evalúa (f: función, p: población) retorna evaluaciones se
    para i:=1 hasta m hacer
        avi:=f(pi);
    fpara;
    retorna av;
ffunción;

función selecciona_mejor (av:evaluaciones,p:población)
    retorna cromosoma se
        mejor:=p1; av_mejor:=av1;
        para i:=2 hasta m hacer
            si (av_mejor ≤ avi) entonces mejor:=pi; av_mejor:=avi; fsi;
    fpara;
    retorna mejor;
ffunción;
```

Quedan pendientes las operaciones de construir la nueva población, que se considera más adelante, y la función que decide si se tiene que acabar el proceso de iteración. Para saber si el proceso tiene que terminar, podemos tener en cuenta el número de iteraciones que queremos hacer (si *k* sobrepasa un de-

terminado valor) o si la solución converge: si la diferencia entre las evaluaciones de dos soluciones consecutivas más buenas es suficientemente pequeña $\| m^{(k)} - m^{(k-1)} \| < \varepsilon$ para un ϵ dado).

Para construir la nueva población a partir de la población actual hay muchas alternativas. A continuación, se muestra una en la que la construcción está dividida en dos etapas. Primero se determinan los cromosomas que entran a formar parte de la nueva población y después se construirá la nueva población operante con los cromosomas (haremos entrecruzamientos y mutaciones de los cromosomas que intervienen).

```
función crea_nueva_población (av:evaluaciones,p:población)
                            retorna evaluaciones se
    total_av:=0;

    para i:=1 hasta m hacer
    total_av:=total_av+avi;
    fpara;

    pr_acc0:=0;

    para i:=1 hasta m hacer
    pr_acci:=pr_acci-1 + avi/total_av;
    fpara;

    para i:=1 hasta m hacer
        r:=valor aleatorio entre 0 y 1;
        p'i:=selecciona pi tal que pr_acci-1<r≤pr_acci;
    fpara;

    para i:=1 hasta m hacer
    p''i:=genera(p'i,p');
    fpara;

    retorna p'';
ffunción;
```

La función calcula la probabilidad de pasar a formar parte de la nueva población para cada cromosoma. Esta probabilidad se define como $av_i/total\_av$ donde `total_av` es la suma de todas las evaluaciones. Observad que este valor se puede entender como una probabilidad porque $\sum_{j \leq i} av_i/total\_av = 1$ y, además, tiene sentido, puesto que la probabilidad es mayor cuanto más bien evaluado ha sido el cromosoma. Aun así, en lugar de guardar la probabilidad, guardamos los acumulados de las probabilidades (lo cual está en pr_acc_i),

puesto que así es más fácil seleccionar después un cromosoma dado un número aleatorio entre cero y uno. Podemos ver que en la función la selección de los nuevos cromosomas se hace primero escogiendo este valor aleatorio y después utilizándolo para seleccionar el cromosoma p_i tal que el valor aleatorio está entre $\texttt{pr\_acc}_{\texttt{i-1}}$ y $\texttt{pr\_acc}_{\texttt{i}}$. Esto se hace m veces con el fin de obtener una población con m cromosomas.

Ahora tenemos que modificar los cromosomas seleccionados para que la nueva población no sea como la anterior, y que haya un **intercambio genético** entre los diferentes individuos. Así, a cada cromosoma de la población seleccionada (p') le aplicamos la función $\texttt{genera(p'}_{\texttt{i}}\texttt{,p')}$ que nos construye el nuevo cromosoma a partir del seleccionado y de la población p'. Para hacer esta construcción hay varias alternativas. Aquí consideramos cuatro tipos de operaciones para conseguir el intercambio genético:

- **Entrecruzamiento simple** (intercambia los genes de un cromosoma con los de otro cromosoma a partir de una posición dada).

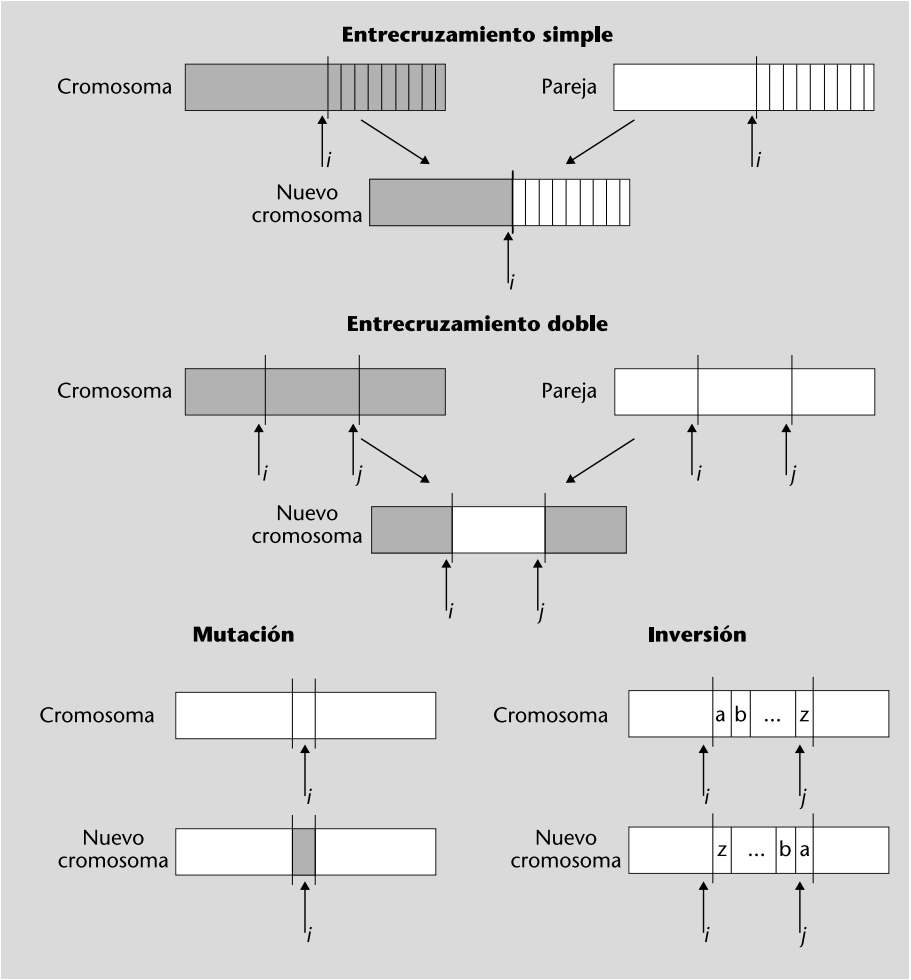
- **Entrecruzamiento doble** (intercambia una parte de los genes de un cromosoma con los de otro cromosoma).

- **Mutación** (un determinado gen de un cromosoma es sustituido por otro gen del *pool*).

- **Inversión** (en un cromosoma, una subsecuencia de genes es invertida).

A continuación se muestran las representaciones gráficas y las implementaciones de estas operaciones:

Figura 8




```
función entrecruzamiento_simple (c:cromosoma,p:población)
                              retorna cromosoma se
    pareja:=selecciona cromosoma de la población(p);
    i:=selecciona posición entre 1 y m;
    para k:=i+1 hasta m hacer
        ck:=parejak;
    fpara;
    retorna c;
ffunción;

función entrecruzamiento_doble (c:cromosoma,p:población)
                               retorna cromosoma se
    pareja:=selecciona cromosoma de la población(p);
    i:=selecciona posición entre 1 y m;
    j:=selecciona posición entre i y m;
    para k:=i+1 hasta j hacer
        ck:=parejak;
    fpara;
    retorna c;
ffunción;
```

```
función mutación (c:cromosoma) retorna cromosoma se
    i:=selecciona posición entre 1 y m;
    g:=selecciona gen (G);
    ci:=g;
    retorna c;
ffunción;

función inversión (c:cromosoma) retorna cromosoma se
    i:=selecciona posición entre 1 y m;
    j:=selecciona posición entre i y m;
    para k:=i+1 hasta j hacer
        ck:=c(i+1)+(j-k);
    fpara;
    retorna c;
ffunción;
```

A partir de estas funciones se puede definir la función `genera` que había quedado pendiente. La función elegirá para cada cromosoma una de las funciones de modificación genética de manera aleatoria. La elección se basa en probabilidades de escoger cada una de las operaciones (las denominamos `pr_entrecruzamiento_simple`, `pr_entrecruzamiento_múltiple`, `pr_mutación`, `pr_inversión`). La suma de estas probabilidades tiene que ser menor o igual que 1 (si la suma de estas probabilidades es α y es menor que 1 querrá decir que un cromosoma no será modificado con probabilidad 1 – α). La definición de la función es la que presentamos a continuación:

```
función genera (c:cromosoma, p:población) retorna cromosoma se
    constantes pr_entrecruzamiento_simple, pr_entrecruzamiento_múltiple;
    constantes pr_mutación, pr_inversión;
    r:=valor aleatorio entre 0 y 1;
    si r ≤ pr_entrecruzamiento_simple entonces
        retorna entrecruzamiento_simple(c,p);
    si no si r ≤ pr_entrecruzamiento_simple+pr_entrecruzamiento_múltiple entonces
        retorna entrecruzamiento_doble(c,p);
    si no si r ≤ pr_entrecruzamiento_simple+pr_entrecruzamiento_múltiple+pr_mutación entonces
        retorna mutación(c);
    si no si r ≤ pr_entrecruzamiento_simple+pr_cruzamiento_múltiple+pr_mutación
             +pr_inversión
    entonces
        retorna inversión(c);
    si no retorna c;
    fsi;
ffunción;
```

5. Partición de datos y protocolos de validación

Cuando se lleva a cabo un proceso de aprendizaje supervisado, especialmente en problemas de clasificación, es importante establecer un protocolo para validar hasta qué punto el sistema ha aprendido bien un conjunto de datos.

La validación más simple, conocida como **validación simple**, consiste en dividir el conjunto de datos en dos: uno denominado de entrenamiento y el otro de test. Se construye el modelo del conjunto de entrenamiento y, a continuación, se clasifican los ejemplos de test con el modelo generado a partir del de entrenamiento. Una vez obtenidas las predicciones, se aplica alguna de las medidas de evaluación como las que se describen en el subapartado siguiente. Una cuestión importante que hay que tener en cuenta cuando dividimos un conjunto de datos en los conjuntos de entrenamiento y test es mantener la proporción de ejemplos de cada clase en los dos conjuntos.

En el supuesto de que tengamos un algoritmo de aprendizaje que necesita hacer ajustes de parámetros, se divide el conjunto de entrenamiento en dos subconjuntos: uno de entrenamiento propiamente dicho y otro de validación. Se ajustan los parámetros entre el conjunto de entrenamiento y el conjunto de validación y, una vez encontrados los óptimos, juntamos el conjunto de entrenamiento con el de validación y generamos el modelo con el algoritmo de aprendizaje. La validación la hacemos del conjunto de test. Un error de método bastante usual es utilizar el conjunto de test para hacer el ajuste de los parámetros. Esto no es estadísticamente correcto. Una de las técnicas más usadas para ajustar los parámetros son los algoritmos de optimización, como, por ejemplo, los algoritmos genéticos, que acabamos de estudiar en el apartado anterior.

Un problema que se ha detectado en el uso de la validación simple es que según el conjunto de datos que tengamos puede variar mucho el comportamiento del sistema en función de la partición de ejemplos que hayamos considerado. Es decir, diferentes particiones de ejemplos conducen a distintos resultados. En muchos casos, los investigadores dejan disponibles los conjuntos ya divididos en entrenamiento y test para que las comparaciones entre sistemas sean más fiables.

Para minimizar este efecto, se suele utilizar la **validación cruzada**[4] en lugar de la simple. Este tipo de validación consiste en dividir un conjunto en *k* subconjuntos.

(4)En inglés, *cross-validation* o *k-fold cross-validation*.

(5)En inglés, *leave-one-out*.

A continuación, se hacen *k* pruebas utilizando en cada una un subconjunto como test y el resto como entrenamiento. A partir de ahí, se calcula la media y la desviación estándar de los resultados. Así podemos ver mejor el comportamiento del sistema. Un valor muy utilizado de la *k* es 10. Una variante conocida de la validación cruzada es **dejar uno fuera**[5]. Este caso es como el anterior: definir la *k* como el número de ejemplos del conjunto total. Nos quedaría el conjunto de test con un único ejemplo. Este método no se suele utilizar debido a su alto coste computacional.

Uno de los problemas que nos encontramos cuando describimos las medidas de evaluación es la gran diversidad de medidas que se utilizan en las diferentes áreas de investigación. Estas medidas suelen tener en cuenta las diferentes peculiaridades de los problemas de los diferentes campos. Otro problema es que a veces reciben nombres diferentes en función del área. En este apartado, veremos las medidas que se utilizan en los contextos de la clasificación, la recuperación de la información y la teoría de la detección de señales.

La mayoría de las medidas de evaluación se pueden expresar en función de la matriz de confusión o tabla de contingencia. La matriz de confusión contiene una partición de los ejemplos en función de su clase y predicción. La tabla 7 muestra el contenido de las matrices de confusiones para el caso binario. A modo de ejemplo, la celda *positivo verdadero* corresponde al número de ejemplos del conjunto de test que tienen tanto la clase como la predicción positivas. Los *falsos positivos* también se conocen como *falsas alarmas* o *error de tipo I*; los *falsos negativos* como *error de tipo II*; los *positivos verdaderos* como *éxitos*; y los *negativos verdaderos* como *rechazos correctos*.

Tabla 7. Matriz de confusión

		Clase real	
		Positiva	Negativa
Predicción	Positiva	Positivo verdadero (*tp*)	Falso positivo (*fp*)
	Negativa	Falso negativo (*fn*)	Negativo verdadero (*tn*)

El **error** o **ratio de error** mide el número de ejemplos que se han clasificado incorrectamente del total de ejemplos.

Se pretende que los algoritmos de aprendizaje tiendan a minimizar esta medida. La fórmula es:

$$\text{error} = \frac{fp + fn}{tp + fp + tn + fn}$$

La **exactitud**[6] corresponde a los ejemplos que se han clasificado correctamente del total de ejemplos. Esta medida es complementaria de la anterior. La fórmula corresponde a:

(6)*Accuracy*, en inglés.

$$\text{exactitud} = \frac{tp + tn}{tp + fp + tn + fn}$$

La **precisión** o **valor predictivo positivo**[7] corresponde a los ejemplos positivos bien clasificados del total de ejemplos con predicción positiva. La fórmula es:

(7)*Precision*, en inglés.

$$\text{precisión} = \frac{tp}{tp + fp}$$

La **sensibilidad**[8] corresponde a los ejemplos positivos bien clasificados del total de ejemplos positivos. La fórmula es:

(8)*Recall*, en inglés.

$$\text{sensibilidad} = \frac{tp}{tp + fn}$$

El conjunto de precisión y sensibilidad se puede ver como una versión extendida de la exactitud. Estas medidas provienen del campo de la recuperación de la información, donde el valor de *tn* es muy superior al resto y sesga las medidas de evaluación.

De ahí mismo proviene la medida *F1*, propuesta en 1979 por Van Rijsbergen, que combina las dos anteriores:

$$F1 = 2 \times \frac{\text{precisió} \times \text{sensibilidad}}{\text{precisión} + \text{sensibilidad}} = \frac{2 \times tp}{2 \times tp + fn + fp}$$

Esta medida descarta los elementos negativos debido al sesgo producido por la diferencia entre el número de ejemplos negativos y positivos. En el ámbito de la recuperación de información llega a ser muy crítico.

Hay otras medidas que se utilizan a partir de la matriz de confusión, como la **especificidad**, que se suelen utilizar en el área de la teoría de detección de señales:

$$\text{especificidad} = \frac{tn}{fp + tn}$$

Todas las medidas de este apartado se han descrito a partir del problema binario, pero también se pueden utilizar en problemas multiclase.

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.). Nueva York: John Wiley and Sons, Inc.

Frank, A.; Asuncion, A. (2010). *UCI Machine Learning Repository* [en línea]. Irvine, CA: University of California, School of Information and Computer Science. [Fecha de consulta: 28 de marzo de 2019]. <http://archive.ics.uci.edu/ml>

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

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