vuela

andaluciavuela.es

CONECTA CON ANDALUCÍA

MACHINE LEARNING

PLATAFORMA DIGITAL DE ANDALUCÍA

Junta de Andalucía

ÍNDICE DE CONTENIDO

1. Introducción a Machine Learning

La **Inteligencia Artificial (IA),** pese a estar en boca de todo el mundo hoy día, es una disciplina muy reciente cuya creación se remonta a los años 50 del s. XX. Según la AAAI (*American Association for Artificial Intelligence*), la IA es una "*disciplina científico-técnica que se ocupa de la comprensión de los mecanismos subyacentes en el pensamiento y la conducta inteligente y su incorporación en las máquinas*". Hay diferentes formas de abordar esta definición a través de la Ingeniería:

Sistemas que piensan como humanos	**Sistemas que piensan racionalmente**
Automatización de toma de decisiones, resolución de problemas, aprendizaje.	Estudio de modelos computacionales de percepción, razonamiento y actuación.
Sistemas que actúan como humanos	**Sistemas que actúan racionalmente**
Hacer que un computador realice tareas sólo atribuibles a humanos.	El desarrollo de comportamientos inteligentes en agentes.

A modo orientativo, la IA tiene como objeto de estudio diferentes áreas:

Representación y elicitación del conocimiento

Modelos lógicos e imprecisos, minería de datos, web semántica, etc.

- **Búsqueda**
  Inteligencia en juegos, búsqueda de soluciones en problemas de logística, GPS, etc.

- **Planificación**
  Plantas industriales, robots autónomos, electrodomésticos inteligentes, domótica, etc.

- **Sistemas Expertos y de soporte a la decisión**
  Diagnóstico médico, asistentes software, agentes de recomendación, buscadores inteligentes.

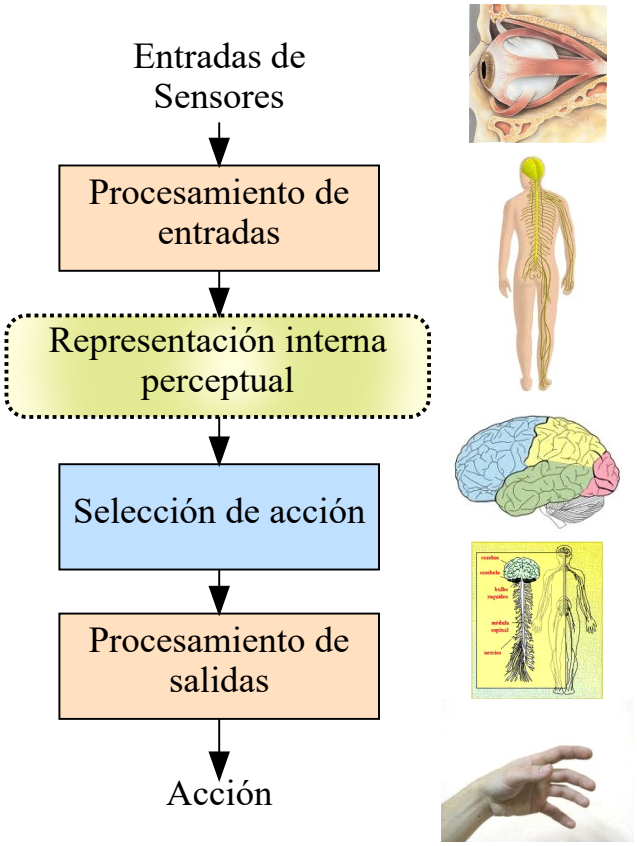
- **Reconocimiento del lenguaje natural**
  Interfaces hombre-máquina, agentes conversacionales, etc.

- **Visión por computador**
  Videovigilancia, procesamiento de imágenes (médicas, satélite), gestión de la calidad alimentaria, etc.

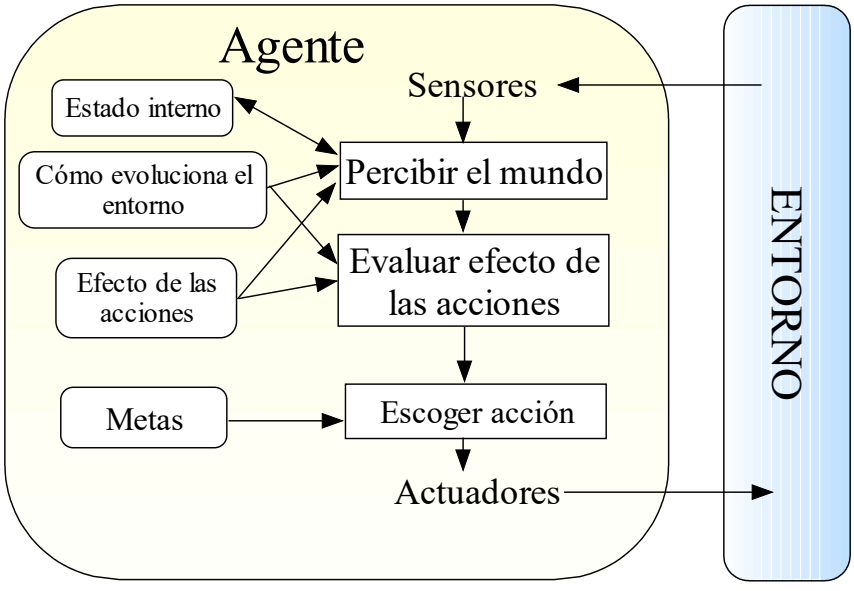
- **Robótica**
  Electrodomésticos inteligentes, plantas industriales, sistemas de interacción hombre-máquina, etc.

- **Computación ubicua e inteligencia ambiental**
  Internet de las cosas, espacios marcados, servicios móviles, etc.

En términos generales, a cualquier sistema (**hardware o software**) que presente inteligencia, se le conoce con el término técnico de **agente (inteligente)** dentro de la rama de la IA. El comportamiento de un agente se rige por el conocido ciclo de **percepción-acción**:



De forma genérica, el diseño interno de un agente inteligente (con posibles modificaciones/adaptación según la aplicación o problema a resolver), es el siguiente:



El **aprendizaje**, como parte integral de una entidad inteligente cubre una amplia gama de fenómenos como:

- El perfeccionamiento de la habilidad.
- La adquisición del conocimiento.

El aprendizaje modifica el mecanismo de decisión del agente para mejorar su comportamiento. Dentro del estudio de la Inteligencia Artificial, encontramos el **aprendizaje automático (Machine Learning, ML)** como una sub-área que persigue el desarrollo de modelos, métodos y algoritmos que permitan a un agente inteligente mejorar su comportamiento con la experiencia. Dentro del ML encontramos 3 vertientes principales o tipos de aprendizaje estudiados:

- **Aprendizaje supervisado:** Se trata de entrenar un modelo o agente para que realice una tarea, dando como datos de aprendizaje patrones (entrada, salida) o **datos etiquetados**. Estos patrones indicarán al modelo del agente que "*ante esta situación (entrada), la forma correcta de comportarse es aquella (salida)*". Con un número suficientemente alto de ejemplos, y el modelo y método de aprendizaje adecuados, el agente aprenderá a partir de la experiencia a mejorar su conducta con respecto a alguna medida de evaluación. Ejemplos de aprendizaje supervisado son el reconocimiento de la voz, reconocimiento de texto manuscrito, detección de SPAM, etc.

- **Aprendizaje no supervisado:** Se trata de descubrir patrones de comportamiento en **datos no etiquetados** (patrones que solamente contienen entradas); es decir, de dotar al agente de capacidad de abstracción suficiente como para descubrir reglas o características del problema abstractas. Ejemplos de aprendizaje no supervisado sería la inferencia de reglas de asociación a partir de los tickets generados en un supermercado (por ejemplo, el 97% de las personas que compran leche también compran yogures), o perfiles de usuario (tipos de clientes de una empresa que tienen unas preferencias/gustos u otros).

- **Aprendizaje por refuerzo:** En este tipo de aprendizaje no existen conjuntos de datos dados como patrones a aprender. Al contrario, el agente interactúa con el entorno directamente, y tratará de aprender a través de su propia experiencia con el mismo. Por ejemplo: Un programa que aprende a jugar al ajedrez, un robot que aprende a moverse por sí mismo, etc.

Tanto en el aprendizaje supervisado como en el no supervisado, también encontraremos diferentes tipos de estrategias o metodologías para abordar un problema:

- **Métodos basados en modelos:** representan el conocimiento aprendido en algún lenguaje de representación (modelo o hipótesis).
- **Métodos basados en instancias:** representan el conocimiento aprendido como un conjunto de prototipos descritos en el mismo lenguaje usado para representar la evidencia.

Actualmente, el aprendizaje automático se usa con diversas finalidades:

- Implementación de tareas difíciles de programar (reconocimiento facial, reconocimiento de la voz, etc.).
- Creación de aplicaciones auto-adaptables (interfaces inteligentes, marketing personalizado, sistemas de recomendación…)
- Minería de datos (análisis de datos inteligente).

Es en esta última aplicación en la que nos centramos en este curso, principalmente.

1.1. Scikit-Learn

Scikit-Learn (SkLearn) es un paquete software que se proporciona para Python como un conjunto de bibliotecas. Está basado en la biblioteca **SciPy** de cálculo científico, e incorpora numerosos métodos de índole generalista de aprendizaje automático con el fin del tratamiento de datos para Data Mining. Por este motivo, y también por su simplicidad de uso y eficiente implementación, se ha convertido en una de las herramientas básicas en la Ciencia de Datos y la IA contemporáneas. La instalación de **SkLearn** se realiza de forma sencilla, escribiendo por línea de comandos:

conda install scikit-learn

o bien:

pip install scikit-learn

Entre las características más relevantes de scikit-learn para el curso, destacamos:

- Incorporación de diversos modelos de ML.
- Incorporación de diversos métodos de aprendizaje para ML.
- Capacidad para desarrollo tanto de aprendizaje supervisado como no supervisado.
- Utilidades para pre-procesamiento de datos.
- Utilidades para selección de características, o datos relevantes para la resolución de un problema.
- Utilidades para realizar estudios de evaluación y validación de modelos aprendidos.
- Incorporación de conjuntos de datos de prueba que facilitan el aprendizaje de la biblioteca.

2. Proceso de desarrollo en Machine Learning

Como se ha comentado previamente, los métodos de ML contemporáneos están muy orientados a la minería de datos y, por tanto, al tratamiento estadísticos de datos. En este sentido, existe una metodología muy bien establecida que hay que seguir en el proceso de construcción de cualquier sistema de ML para análisis de datos:

- **Obtención de datos.** Identificación de patrones y su tipo (patrones Entrada/Salida, patrones Entrada…).
- **Pre-procesamiento**. Tratamiento inicial de los datos, incluyendo:
  - Gestión de valores perdidos.
  - Gestión de conjuntos de datos no balanceados.
  - Transformaciones de los datos.
- **Selección de características.** Entre todos los atributos que conforman los patrones de entrada, seleccionar aquellos (o combinaciones de aquellos) que sean los más relevantes para solucionar el problema, descartando el resto.
- **Elección del modelo de comportamiento.** Selección del modelo matemático que regirá el comportamiento del agente (árboles de decisión, redes neuronales, sistemas de reglas…).
- **Elección del mecanismo de aprendizaje.** Escoger, entre las técnicas de aprendizaje para el modelo existentes, la más adecuada, junto con el criterio de evaluación de la actuación del agente.
- **Aprendizaje y ajuste de hiperparámetros.** Entrenamiento del modelo con los datos, mediante el mecanismo de aprendizaje seleccionado.
- **Validación.** Comprobación de que la abstracción aprendida por el agente es apropiada (no es tan genérica que no aprende, ni tan especifica que sólo ha aprendido los datos que se le ha entregado para entrenarlo).

3. Modelos y algoritmos

En Machine Learning, independientemente del modelo de aprendizaje a utilizar, existen unos conceptos comunes a todo:

- **Modelo**: Los modelos serán implementaciones de sistemas matemáticos formales, que indicarán las reglas del comportamiento del agente inteligente para la toma de decisiones. Se notará habitualmente como una función **f** que proporcione **y'=f(x,w)** (abreviado como **y=f(x)**); donde **x** es un patrón de entrada, **y'** es la salida del modelo (la decisión esperada del agente inteligente), y **w** son parámetros de **f** que deben ajustarse para que **f** tenga el comportamiento deseado. Por ejemplo, suponiendo que queremos representar una regresión lineal, entonces **f(x,w)= w_0 + w_1*x**. Algunos modelos comunes son las redes neuronales artificiales, los árboles de decisión, los sistemas de reglas, etc.

- **Algoritmo de aprendizaje**: Los modelos por sí mismos no "nacen" o "se implementan" con el comportamiento deseado. Usualmente, estos modelos contienen parámetros que hay que ajustar para que el agente realice un desempeño de la actividad satisfactorio. Los algoritmos de aprendizaje sirven para dos motivos principales:

    - **Inferencia de modelos**: Encontrar el modelo óptimo que desempeña correctamente una actividad. Por ejemplo: Encontrar un sistema de reglas apropiado que consiga hacer funcionar un ascensor correctamente.
    - **Estimación de parámetros**: Encontrar cuáles son los mejores parámetros **w** posibles para que un modelo **f(x,w)**

Estimación de hiperparámetros: Se considera un hiperparámetro a los parámetros genéricos que rigen el comportamiento tanto de modelos como de algoritmos de aprendizaje. Por ejemplo, en modelos de redes neuronales artificiales un hiperparámetro habitual es el número de neuronas o su función de activación. Otro hiperparámetro habitual (en este caso de algoritmos de aprendizaje) es la tasa de aprendizaje, que indica cómo de rápido deben de ajustarse los parámetros del modelo para llegar al comportamiento deseado.

Entrenamiento de modelos ya construidos (optimización de parámetros)

A modo ejemplo, implementaremos un modelo de neurona artificial tipo **Perceptrón**, y lo usaremos para clasificación. El perceptrón se estudiará en mayor detalle en el próximo módulo, aunque su funcionamiento es el siguiente:

$$y' = f(z)$$

$$z = b + WX$$

Donde **X es un vector** de atributos numéricos de un patrón, **W** es una matriz de parámetros y **b** otro parámetro denominado ***bias*. El término f** se denomina ***función de activación*** y, en la implementación que realizamos, será:

$$y' = \begin{cases} 1 & z \geq 0 \\ -1 & z < 0 \end{cases}$$

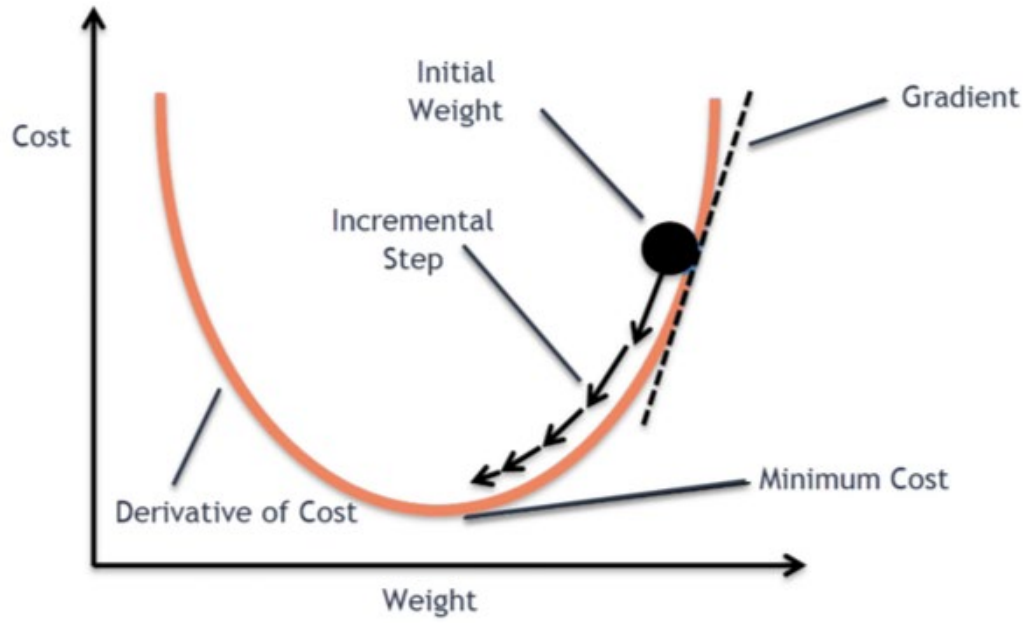
Ejemplo: Implementación de un modelo perceptrón

Ver ficheros Python asociados: **Codigo1.py**

Como se decía anteriormente, los **algoritmos de aprendizaje** son métodos de extracción de modelos a partir de datos, o métodos de optimización numérica que permiten encontrar los parámetros óptimos de un modelo. Para ello, es necesario indicar qué es un valor óptimo, y ello conlleva la **minimización de una función de error**, también denominada **función de coste o función de pérdida**. En nuestro caso, como deseamos que el perceptrón proporcione unas salidas deseadas, lo que se busca es ***minimizar el error entre las salidas del modelo y las salidas reales*** que usaremos para entrenarlo. Si tenemos un conjunto de **N** patrones de entrada/salida **$\{(x_i, y_i)\}$**, una función de coste habitual es el error cuadrático entre las salidas esperadas y las salidas del modelo:

$$J(w) = \frac{1}{2}\sum_{i=1}^{N}(y_i - f(x_i, w))^2$$

Los métodos de minimización de este tipo de funciones proceden el cálculo numérico, son algoritmos iterativos y están basados en el cálculo de la primera o la segunda derivada, según la dirección del gradiente. La idea es ir actualizando iterativamente los parámetros **w** del modelo, de modo que se minimice la función de coste:



Esta actualización, como se ve en la figura, debe realizarse en el sentido opuesto de la dirección del gradiente con la siguiente fórmula:

$$w = w + \Delta w$$

$$\Delta w = -\lambda \nabla J(w)$$

En el caso del modelo perceptrón propuesto:

$$\nabla J(w) = \frac{d\,J}{d\,w_j} = -\sum_{i=1}^{N} \left(y_i - f(z_i)\right) x_i^j$$

Este método se conoce como **Descenso del Gradiente (Gradient Descent**) y tanto él como sus variantes son una parte esencial del aprendizaje en Deep Learning.

Inferencia de modelos

En el caso anterior, construíamos un modelo fijo a priori, con parámetros que posteriormente había que optimizar (fase de entrenamiento/aprendizaje). En el caso de la **inferencia de modelos**, no existe un modelo fijo creado a priori, sino que un algoritmo se encarga de construirlo (estos modelos no suelen tener parámetros a optimizar posteriormente). Es el caso de los árboles de decisión, o de problemas de regresión simbólica, por ejemplo.

A título ilustrativo, consideremos la siguiente tabla, donde aparece un conjunto de datos sobre la decisión de jugar o no jugar al golf dependiendo de condiciones atmosféricas como el aspecto del cielo, la temperatuda, la humedad y el viento. Podríamos generar un sistema experto de IA que automáticamente decida si se va a jugar o no en base a los datos aprendidos. El modelo en particular que decidimos usar es un **árbol de decisión**.

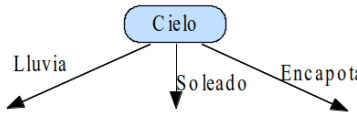
La construcción de este tipo de modelos se realiza a partir de algoritmos de extracción de árboles de decisión, como son el ID3 o el C4.5, por mencionar dos métodos clásicos. Como hiperparámetro, se suele tener una métrica o medida que pérmita calcular la información que aporta cada atributo para la predicción de la variable a calcular deseada, como por ejemplo la entropía de la información:

$$H(X) = -\sum_{i=1}^{n} p\,(x_i) log p(x_i)$$

Dia	Cielo	Temp.	Humedad	Viento	Jugar
1	Soleado	Calor	Alta	Débil	No
2	Soleado	Calor	Alta	Fuerte	No
3	Encapotado	Calor	Alta	Débil	Si
4	Lluvia	Suave	Alta	Débil	Si
5	Lluvia	Frío	Normal	Débil	Si
6	Lluvia	Frío	Normal	Fuerte	No
7	Encapotado	Frío	Normal	Fuerte	Si
8	Soleado	Suave	Alta	Débil	No
9	Soleado	Frío	Normal	Débil	Si
10	Lluvia	Suave	Normal	Débil	Si
11	Soleado	Suave	Normal	Fuerte	Si
12	Encapotado	Suave	Alta	Fuerte	Si
13	Encapotado	Calor	Normal	Débil	Si
14	Lluvia	Suave	Alta	Fuerte	No

Partiendo de esta medida discriminatoria, se irían creando nodos de un árbol, asociados a atributos, y arcos asociados a valores de estos atributos hasta que no hubiese más información para, finalmente, llegar a los nodos hoja que contienen la decisión a tomar.

Suele ser un proceso recursivo que parte de un conjunto de datos completo, que se va dividiendo según la medida de información escogida.



Cielo=Lluvia

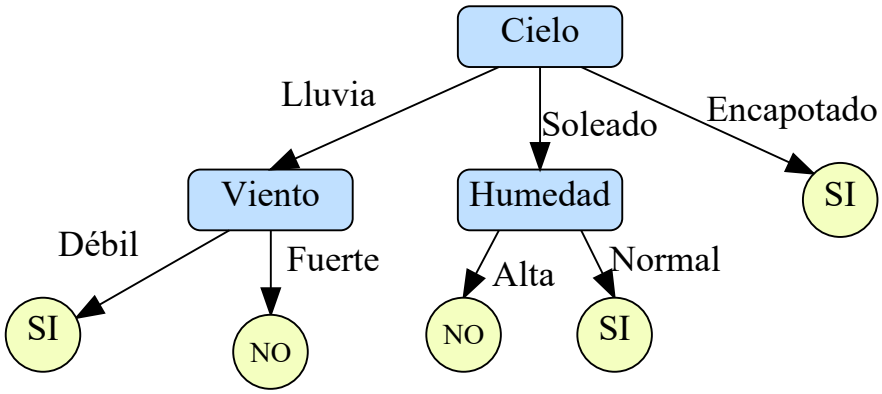
Temp	Humd.	Viento	Jugar
Suave	Alta	Débil	Si
Frío	Normal	Débil	Si
Frío	Normal	Fuerte	No
Suave	Normal	Débil	Si
Suave	Alta	Fuerte	No

Cielo=Encapotado

Temp	Humd.	Viento	Jugar
Calor	Alta	Débil	Si
Frío	Normal	Fuerte	Si
Suave	Alta	Fuerte	Si
Calor	Normal	Débil	Si

Cielo=Soleado

Temp	Humd.	Viento	Jugar
Calor	Alta	Débil	No
Calor	Alta	Fuerte	No
Suave	Alta	Débil	No
Frío	Normal	Débil	Sí
Suave	Normal	Fuerte	Sí



4. Tratamiento inicial de los datos

Como se ha comentado previamente, el aprendizaje automático contemporáneo se centra en la creación de modelos para procesamiento de datos. Estos datos, habitualmente, no suelen estar en un formato apropiado para procesamiento por un modelo de Machine Learning o, aún estándolos, conviene transformarlos o realizar operaciones con ellos a fin de reducir su complejidad. En este sentido, conviene detectar cuál es la **codificación óptima** de los datos, tratar valores perdidos, y realizar transformaciones a los mismos, como paso previo de la selección de las variables de mayor relevancia para el problema.

4.1. Codificación de datos

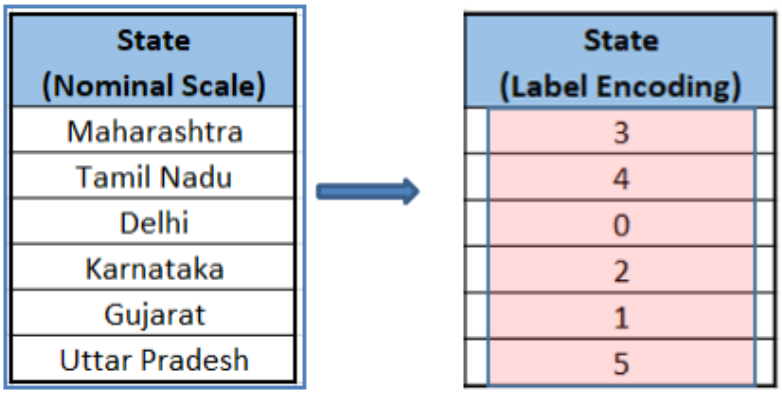
Usualmente, los modelos de ML trabajan con datos numéricos. Por tanto, ante la presentación de datos categóricos (nombres, tipos, colores…), es necesario realizar una correcta codificación numérica de los mismos. Las principales codificaciones que usaremos son:

Codificación real

Todas las variables numéricas continuas (con valores decimales) tienen este tipo de codificación.

Codificación entera (de orden o de etiqueta)

Se corresponde con la asignación de un valor entero {0, 1, 2 …} a un conjunto de valores de una variable categórica. Por ejemplo, si un atributo tiene valores {alto, bajo}, su codificación entera podría darse a partir de la aplicación alto-> 0; bajo -> 1.



State (Nominal Scale)
Maharashtra
Tamil Nadu
Delhi
Karnataka
Gujarat
Uttar Pradesh

State (Label Encoding)
3
4
0
2
1
5

Codificación binaria (one-hot)

Se corresponde con la asignación de un valor de variable categórica a un valor array con todas sus componentes a 0, salvo una que tiene valor 1. Es muy útil para codificar valores categóricos donde no existe una relación de orden entre los elementos.

id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

Ejemplo: Codificación de datos con SKLearn

Ver ficheros Python asociados: **Codigo2.py**

4.2. Transformaciones de datos

Algunas de las transformaciones de datos más usuales son:

- **Cambio de escala:** Cambia el valor de los datos entre un valor mínimo y otro máximo a otra escala (otro valor mínimo/máximo). Clase **MinMaxScaler(feature_range=(min, max))**.
- **Estandarización**: Eliminación de media y varianza (normalización de datos a media 0 y desviación estándar 1). Clase **StandardScaler**, o **RobutsScaler** en el caso de que existan múltiples datos *outliers* (en este caso, se escoge la mediana en lugar de la media, y se usa el rango intercuartílico en lugar de la desviación típica)
- **Otras transformaciones personalizadas: FunctionTransformer**. Implementa la transformación deseada en una función como parte de sklearn.

Ejemplo: Transformaciones de datos
Ver ficheros Python asociados: **Codigo3.py**

4.3. Imputación de valores perdidos

Aunque ***Pandas*** proporciona algunos mecanismos para tratar valores perdidos, ***sklearn*** incorpora también de serie algunas técnicas para su ***imputación*** (cálculo aproximado del contenido de un valor perdido). Las técnicas más usadas de imputación de valores perdidos en ***sklearn*** son:

- Reemplazar valor perdido con la media: **sklearn.impute.SimpleImputer(strategy='mean')**.
- Reemplazar valor perdido con la mediana: **sklearn.impute.SimpleImputer(strategy='median')**.
- Reemplazar valor perdido con la moda: **sklearn.impute.SimpleImputer(strategy='most_frequent')**.
- Reemplazar valor perdido con un valor constante: **sklearn.impute.SimpleImputer(strategy='constant', fill_value= valor)**.
- Reemplazar valor perdido con valor de los patrones más similares existentes: **sklearn.impute.KNNImputer(n_neighbors=vecinos)**

Ejemplo: Tratamiento de valores perdidos
Ver ficheros Python asociados: **Codigo4.py**

4.4. Selección de características

La selección de características permite escoger, entre todas las variables existentes en el problema, un subconjunto de las mismas que sea relevante y suficiente para realizar el análisis del problema. Gracias a este conjunto de técnicas, se puede reducir el número de variables independientes en un gran número. Entre las técnicas de selección de características, encontramos las más frecuentes como:

- **Eliminación de atributos con baja varianza:** Si los valores de un atributo son más o menos constantes, entonces el atributo no es significativo (muchos patrones tendrán el mismo valor, y no servirá para discriminar patrones). Clase **sklearn.feature_selection.VarianceThreshold**.

- **Selección de los K mejores atributos** con respecto a una métrica de evaluación dada. Clase **sklearn.feature_selection.SelectKBest**.

- **Eliminación recursiva de atributos poco representativos**. Se aplica un algoritmo voraz que, iterativamente, va eliminando algunos atributos que son poco significativos con respecto a la resolución del problema. Clase **sklearn.feature_selection.RFE**.

- **Análisis de Componentes Principales (PCA)**. Consiste en realizar una transformación lineal a los datos, de modo que el conjunto de datos resultante ordene los nuevos atributos de más a menos representativo. Clase **sklearn.decomposition.PCA**.

Ejemplo: Selección de características con sklearn

Ver ficheros Python asociados: **Codigo5.py**

5. Aprendizaje Supervisado

Como se ha comentado previamente, el aprendizaje automático contemporáneo se centra en la creación de modelos de comportamiento (modelos matemáticos) con parámetros, que deben ajustarse a través de un proceso de aprendizaje para optimizar el desempeño de la actividad deseada. Actualmente, la mayoría de las técnicas de aprendizaje automático se engloban dentro del **aprendizaje inductivo**, que tiene las siguientes características:

- Se aprende a partir de datos existentes (patrones).
- Existen datos de entrada (patrones de entrada).
- Existen datos de características de salida (patrones de salida, sólo para aprendizaje supervisado).

Los modelos de comportamiento se pueden expresar de diversas formas:

- Árboles de decisión.
- Reglas.
- Redes neuronales.
- Modelos bayesianos o probabilísticos.
- etc.

Los árboles de decisión y la extracción de reglas son algunos de los modelos más usados en aprendizaje automático, debido a su fácil interpretación y posterior validación para uso industrial.

5.1. Clasificación

Uno de los grandes problemas con los que trata la Inteligencia Artificial es el problema de clasificación. Este problema consiste, esencialmente, en encontrar a qué categoría pertenece una muestra recién adquirida. Ejemplos de aplicación son el reconocimiento de objetos en imágenes, la clasificación de calidad de verduras con respecto a sus características, la previsión de concesión de préstamos hipotecarios, categorización de clientes, etc.

La **clasificación binaria** o **dicotómica** es un caso particular del problema de clasificación, donde sólo existe una (o dos) clases a diferenciar, como por ejemplo: detección de intrusos en viviendas, detección de correo SPAM o, en general, una pregunta cuya respuesta pueda ser expresada en términos de "Sí" o "No" (usualmente, usaremos los términos "*positivo*" y "*negativo*".

Un **clasificador** es un modelo matemático que permite, a partir de datos de variables denominadas ***atributos*** o ***variables independientes***, predecir un valor o ***clase/variable dependiente*** asociada a dichos atributos. En argot del aprendizaje automático, también se conocen como **patrones de entrada (variables independientes) y salida (variable dependiente)**. También se suele usar el término **patrón** para referirnos al par (atributos, clase) de un individuo de la población.

Algunos ejemplos de problemas de clasificación son: Determinar si un estudiante es bueno o no en función de sus calificaciones, identificar si el agua contiene algún residuo nocivo para la salud partiendo de sus análisis clínicos, conocer si un paciente sufre cáncer o no a partir de sus análisis clínicos, reconocer un dígito a partir de una imagen de texto manuscrito, etc.

De momento no entraremos en detalle sobre cómo se construye un clasificador, aunque nos centraremos en clasificadores binarios. Modelaremos un clasificador binario como una función $f: X \rightarrow \{positivo, negativo\}$ que, dado como entrada un conjunto de valores de atributos **x** , proporciona el valor **f(x)=positivo** si los atributos se reconocen y se asocian a una clase, y el valor **f(x)=negativo** si se asocian a otra clase.

La principal tarea abordada cuando se soluciona un problema de clasificación reside en ajustar el clasificador para que funcione de forma óptima. Una vez que se ajusta dicho clasificador, debemos comprobar su correcto funcionamiento. Desde el punto de vista estadístico existen múltiples medidas para realizar dicha tarea, aunque comentamos las más comunes:

- **Positivos bien clasificados (*True Positive, TP*).** Indica el número de patrones **{x_i}** en la muestra que deben tener clase positiva y el clasificador devuelve **f(x_i)=positivo.** Cuando el valor se expresa en tanto por uno, se denomina **tasa de positivos bien clasificados (*True Positive Rate, TPR*)**.

$$TPR = \frac{TP}{TP + FN}$$

- **Falsos positivos (*False Positive, FP*).** Indica el número de patrones **{x_i}** en la muestra que deben tener clase negativa y el clasificador devuelve **f(x_i)=positivo.** Cuando el valor se expresa en tanto por uno, se denomina **tasa de falso positivos (*False Positive Rate, FPR*)**.

$$FPR = 1 - TPR$$

- **Negativos bien clasificados (*True Negative, TN*).** Indica el número de patrones **{x_i}** en la muestra que deben tener clase negativa y el clasificador devuelve **f(x_i)=negativo.** Cuando el valor se expresa en tanto por uno, se denomina **tasa de negativos bien clasificados (*True Negative Rate, TNR*)**.

$$TNR = \frac{TN}{TN + FP}$$

- **Falsos negativos (*False Negative, FN*).** Indica el número de patrones **{x_i}** en la muestra que deben tener clase positiva y el clasificador devuelve **f(x_i)=negativo.** Cuando el valor se expresa en tanto por uno, se denomina **tasa de negativos bien clasificados (*False Negative Rate, FNR*)**.

$$FNR = 1 - TNR$$

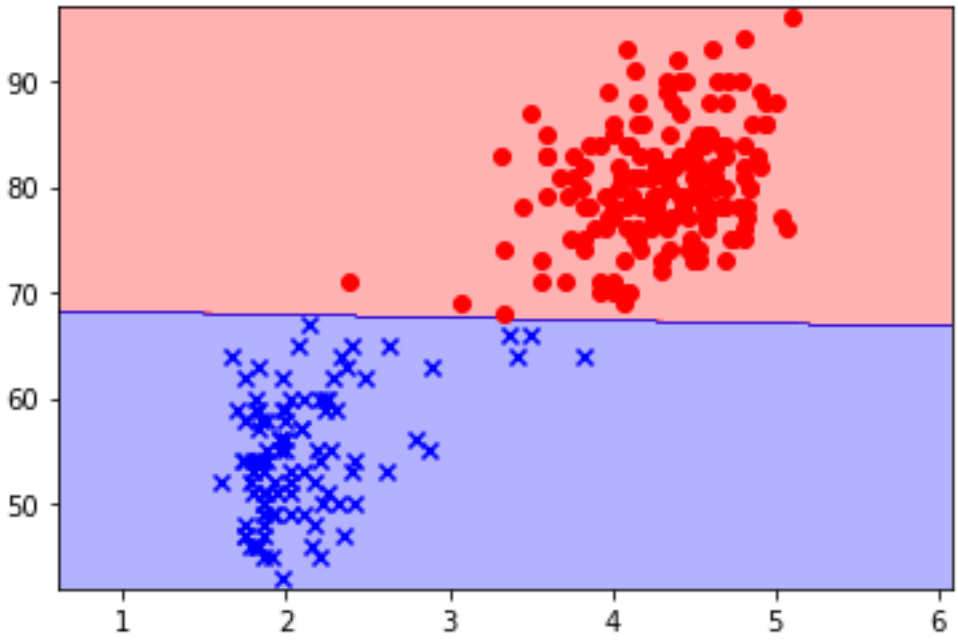
Precisión (*accuracy*): Es la proporción:

$$acc = \frac{TP + TN}{N}$$

Donde **N** es el tamaño de la muestra.

Clasificadores lineales

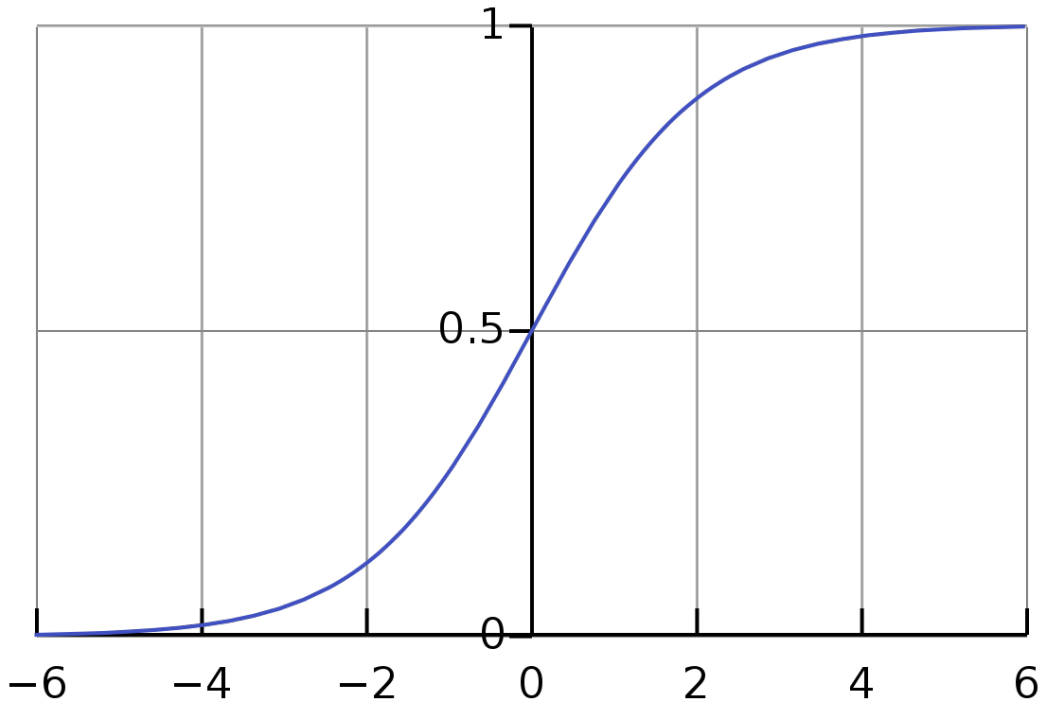
Una de las formas más simples de clasificar es mediante la separación de las clases de un conjunto de datos trazando una recta. De este modo, los patrones que se encuentren a un lado de la recta se clasificarán como positivos, y los del otro lado como negativos.



Uno de los métodos más simples de clasificación con esta técnica consiste en la **regresión logística**. El modelo de regresión logística es:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Donde $x = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$. En este caso, cada $\mathbf{x_i}$ es un atributo numérico, y cada $\mathbf{w_j}$ es un parámetro del modelo, que hay que encontrar. La función logística tiene la siguiente forma:



Por tanto, devolverá valores cercanos a 1 cuando se encuentre la clase positiva, y valores cercanos a 0 cuando sea la clase negativa. La optimización de los parámetros del modelo se realiza mediante la minimización de la siguiente función (**función de pérdida o función de error**):

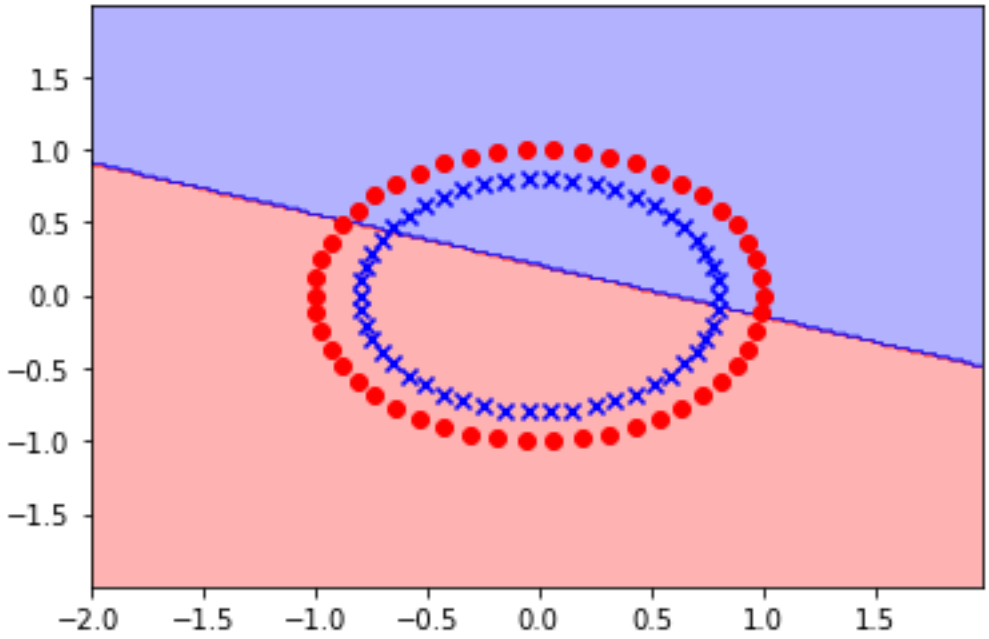
$$w_j^* = \min \{\sum_{i=1}^{N} -y_i \log(y_i') - (1 - y_i)\log(1 - y_i')\}$$

Donde los valores y_i son las salidas esperadas del modelo (valores 0/1), e y_i' son las salidas del modelo, o $y_i'=f(x_i)$, para cada patrón desde el primero (1) hasta el último (**N**).

Ejemplo: Clasificación con modelos de regresión logística

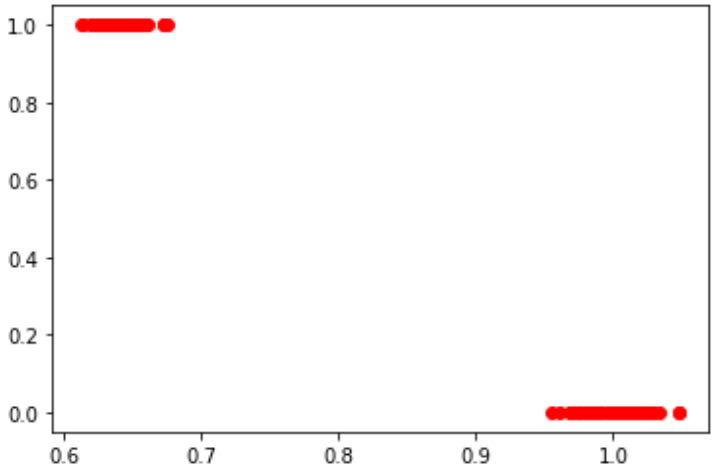
Ver ficheros Python asociados: **Codigo6.py**

La condición de que las dos clases sean **linealmente separables** (que exista al menos un modelo lineal que permita diferenciar las regiones donde se concentran las instancias de cada clase), es usualmente una condición que no se cumple por regla general, salvo en problemas sencillos. Un ejemplo de conjunto de datos con dos clases y 2 atributos de entrada donde no existe separación lineal es el siguiente:



No obstante, en muchos casos es posible expresar los datos de otra forma, mediante transformaciones (no lineales) o proyecciones, que conviertan el conjunto de datos inicial (no linealmente separable) en otro que sea linealmente separable.

En el caso de la figura anterior, basta con agregar los datos calculando la norma de los dos atributos de entrada:



Ejemplo: Clasificación con datos no lineales

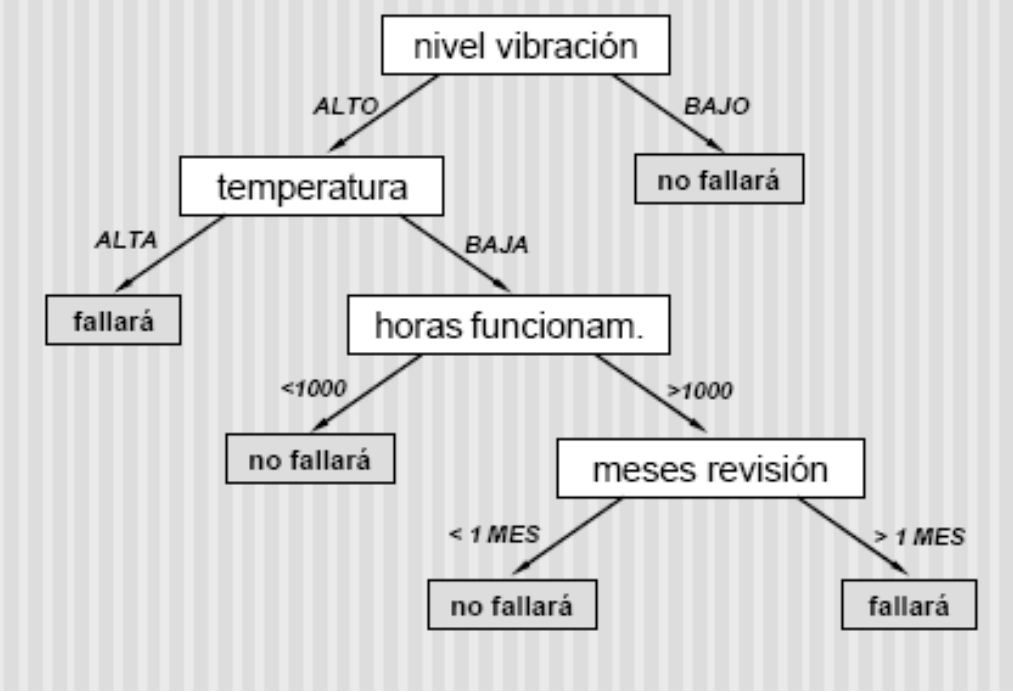
Ver ficheros Python asociados: **Codigo7.py**

Clasificadores por árboles de decisión

Un árbol de decisión es una estructura jerárquica formada por nodos y aristas, donde se distingue un nodo principal o raíz. Cada nodo intermedio del árbol cuenta se asocia a un atributo del conjunto de entrada, y cada arista que sale del nodo hacia sus hijos tiene asociado un valor o conjunto/intervalo de valores. Los nodos hoja (parte más profunda del árbol) tienen asignado un valor de clase, de modo que se puede clasificar un patrón de entrada según sus valores de atributos, observando el árbol. Por ejemplo, para el conjunto de datos siguiente se quiere predecir el atributo (clase) de probabilidad de fallo:

Temperatura	Nivel de vibraciones	Horas de funcionamiento	Meses desde revisión	Probabilidad de fallo
ALTA	ALTO	< 1000	> 1 MES	fallará
BAJA	BAJO	< 1000	< 1 MES	no fallará
ALTA	BAJO	>1000	> 1 MES	no fallará
ALTA	BAJO	< 1000	> 1 MES	no fallará
BAJA	ALTO	< 1000	> 1 MES	no fallará
BAJA	ALTO	>1000	> 1 MES	fallará
ALTA	ALTO	< 1000	< 1 MES	fallará

Un posible árbol de decisión que modela este conocimiento sería:



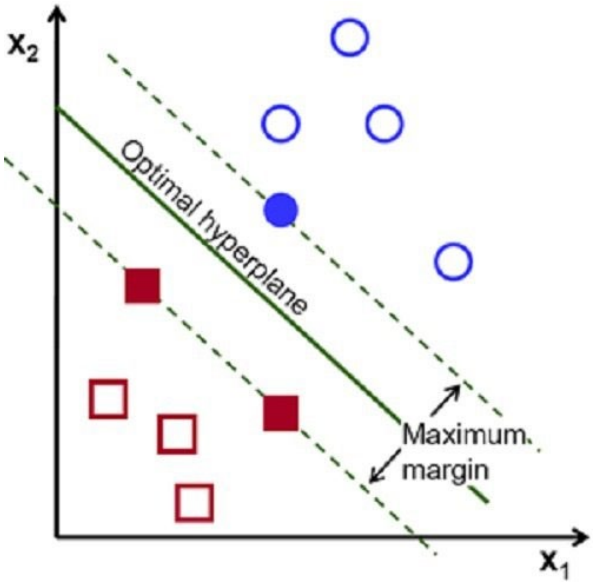
La extracción de árboles de decisión se realiza mediante la separación de atributos **relevantes** en sus respectivos valores, usando algoritmos como **ID3, C4.5**, etc. La decisión de qué medida se debe usar para decidir qué atributos son relevantes y cuáles no suele ser un hiperparámetro, y encontramos medidas como la entropía, el índice de Gini, etc.

Ejemplo: Clasificación con árboles de decisión

Ver ficheros Python asociados: **Codigo8.py**

Clasificadores por máquinas de vectores soporte

Las Máquinas de Vectores Soporte (Support Vector Machines, SVM) son un modelo matemático, basado inicialmente en composición de modelos lineales o ***vectores soporte***, ampliamente utilizados en clasificación. La idea subyacente en las máquinas de vectores soporte para clasificación (***Support Vector Classifiers, SVC***), se basan en encontrar la **línea divisoria de margen máximo**; es decir, aquella línea que divida el espacio de separación de ambas clases de la forma más equitativa posible:



Al igual que en los clasificadores lineales, se requiere que las clases estén etiquetadas con salidas de valores 0/1, de modo que se pueda optimizar la siguiente función de pérdida:

$$f^*(x) = \begin{cases} \max(0,1 - W^T x) & y = 1 \\ \max(0,1 + W^T x) & y = 0 \end{cases}$$

Donde **W** representa una matriz con los parámetros de los diferentes modelos lineales existentes en la SVM.

Ejemplo: Clasificación con SVM

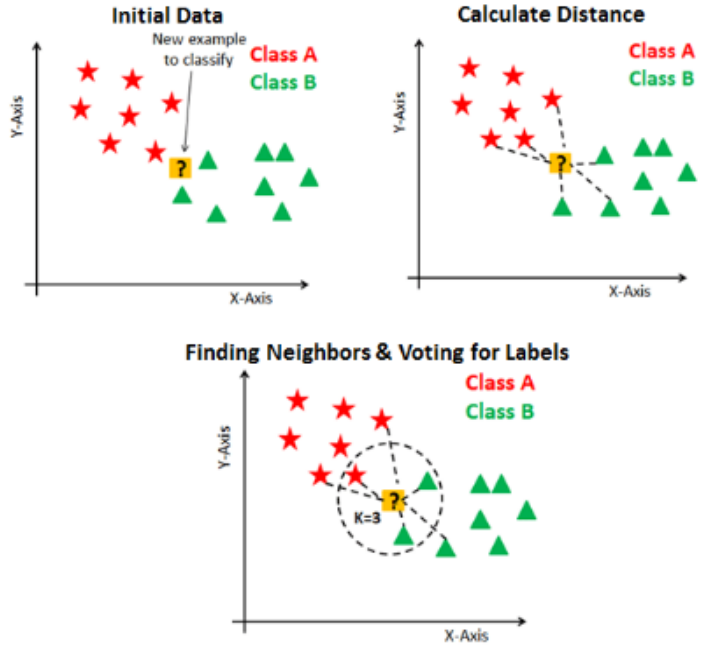
Ver ficheros Python asociados: **Codigo9.py**

Clasificadores por vecino más cercano

Los clasificadores por vecino más cercano requieren una gran cantidad de memoria, para almacenar el conjunto de entrenamiento que se usará para predecir la clase. Su entrenamiento es muy sencillo, dado que sólo requiere un conjunto de datos. Se requiere, además, una función **de comparación de patrones de entrada**, que permita conocer la distancia entre dos patrones de entrada distintos.

El funcionamiento del vecino más cercano es el siguiente:

1. Sea **I** una nueva instancia (patrón de entrada) a clasificar.
2. Se compara **I** con todos los elementos del conjunto de entrenamiento **X**, utilizando una métrica pre-establecida.
3. Se escogen los **k** patrones de **X** más cercanos a **I**, y se anota la clase a la que pertenecen.
4. Se clasifica **I mediante la agregación de las clases de los *k* vecinos más cercanos** (por ejemplo, la clase mayoritaria).

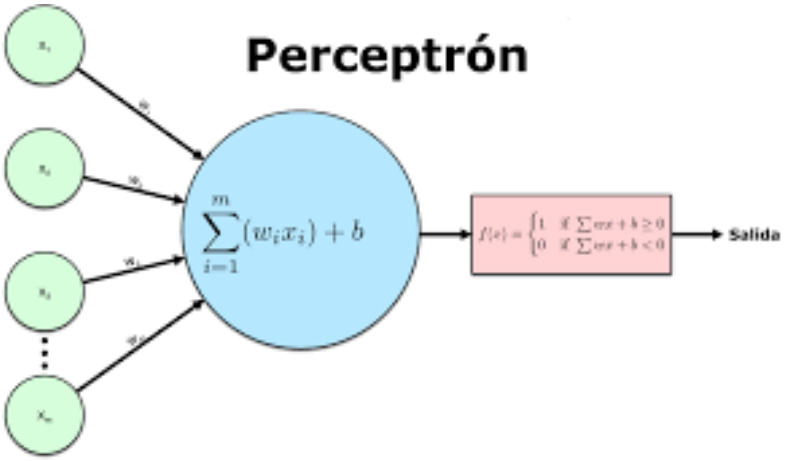


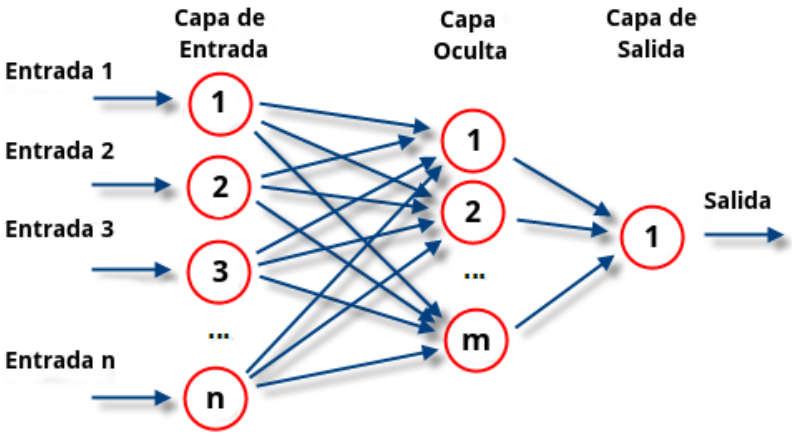
Ejemplo: Clasificación con k-NN

Ver ficheros Python asociados: **Codigo10.py**

Clasificadores por redes neuronales artificiales

Aunque se estudiarán en mayor detalle en el próximo módulo, las redes neuronales artificiales se utilizan con frecuencia para resolver problemas de clasificación. En particular, el modelo más simple (Perceptrón / Perceptrón multicapa) es uno de los métodos más usados. Puede utilizar diferentes funciones de pérdida para entrenar sus parámetros (Error Cuadrático Medio, Entropía Cruzada, etc.





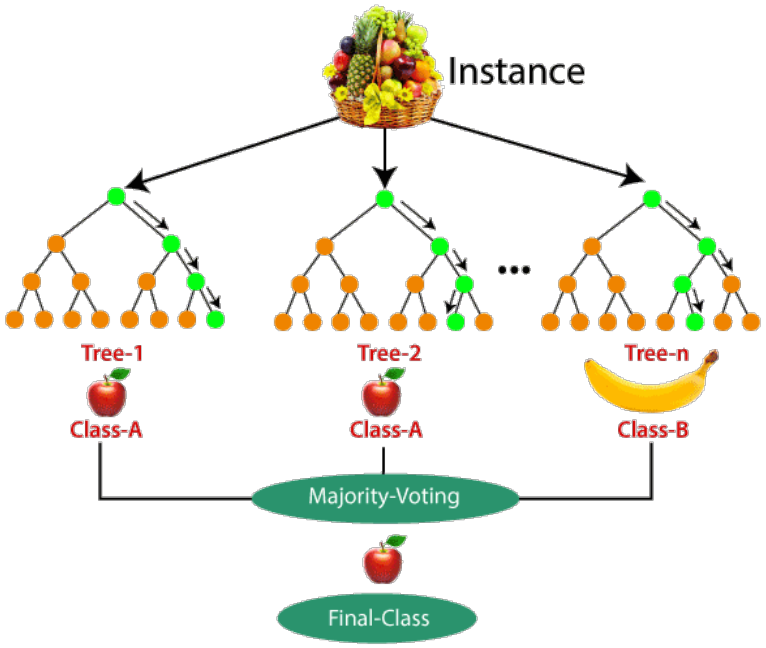
Ejemplo: Clasificación con redes neuronales (Perceptrón)

Ver ficheros Python asociados: **Codigo11.py**

Clasificadores por modelos de agregación (ensemble)

Los clasificadores por modelos de agregación disponen de dos o más clasificadores que predicen la clase a la que pertenece un patrón de entrada. Seguidamente, existe una función de agregación (por ejemplo, la clase más votada) que decide finalmente, en base a la respuesta de todos los clasificadores, cuál clase debe predecirse con el modelo agregado,

En treo los modelos de agregación, uno de los más famosos por su versatilidad, fácil interpretabilidad y robustez, son los **random forests**. Un **Random Forest** está equipado con un conjunto de árboles de decisión diferentes, que han sido entrenados para resolver un problema, junto con una función de agregación que permite discernir cuál respuesta debe proporcionarse, entre todas las que indican los árboles del **random forest**.



Ejemplo: Clasificación con Random Forests

Ver ficheros Python asociados: **Codigo12.py**

5.2. Clasificación con múltiples clases

Aunque la clasificación binaria es uno de los problemas más comunes, también lo es la clasificación donde existen varias categorías o clases. Por ejemplo, el reconocimiento de texto escrito (cada letra/número/signo de puntuación es una clase a clasificar). Algunos modelos, como la regresión logística, pueden adaptarse mediante la estrategia **One-Versus-Rest (OVR)**; es decir, construyendo varios clasificadores (uno por cada clase), y entrenar cada clasificador para aprender a diferenciar los patrones de esa clase frente al resto. Sin embargo, otros modelos sí están nativamente diseñados para poder trabajar con múltiples clases, como son los árboles de decisión, k-NN, redes neuronales artificiales, etc.

En el problema de clasificación multi-clase tenemos los mismos conceptos que en cualquier problema de clasificación; no obstante, las matrices de confusión son de tamaño **CxC** (donde **C** es el número de clases existentes). Además, podemos usar otras métricas para evaluar el desempeño del clasificador, como son:

- **Accuracy**: Porcentaje de patrones bien clasificados frente al total de patrones existentes.
- **Precision**: Se define como la tasa entre positivos bien clasificados frente a todos los elementos que se clasifican como positivos.
- **Recall**: Se define como la tasa entre positivos bien clasificados frente al total de positivos reales existentes en el dataset.
- **F1-Score**: Combina las métricas de Precision y Recall en una única medida.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 - Score = 2\frac{Precision * Recall}{Precision + Recall}$$

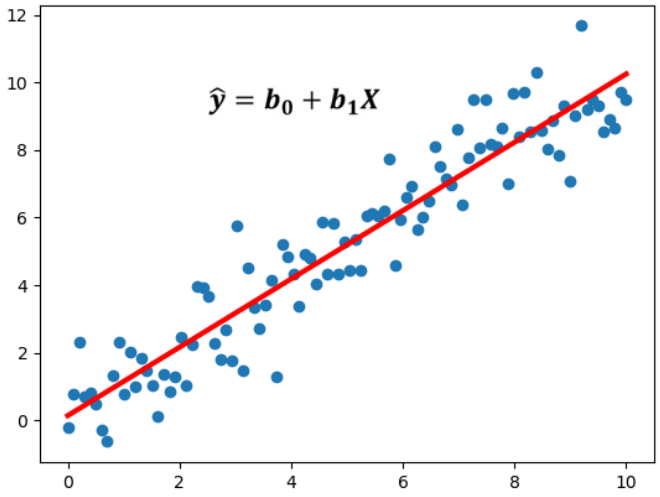
Todas ellas se encuentran disponibles en ***sklearn*** en los paquetes ***sklearn.metrics.accuracy_score, sklearn.metrics.precision_score, sklearn.metrics.recall_score, sklearn.metrics.f1_score***.

Ejemplo: Clasificación multi-clase

Ver ficheros Python asociados: **Codigo13.py**

5.3. Regresión

El problema de regresión, o de aproximación funcional, consiste en la predicción de los valores de una variable numérica continua en función de otras variables numéricas continuas. El caso más común es la regresión lineal:



La práctica totalidad de los modelos previamente estudiados para clasificación también disponen de su correspondiente adaptación para la resolución de problemas de regresión. No obstante, como es lógico, la función de pérdida a optimizar debe ser diferente. Una de las más comunes consiste en minimizar el error cuadrático medio producido entre los patrones de datos de salida reales (**y**) y los proporcionados por el modelo (**y'**):

$$MSE(y, y') = \sum_{i=1}^{N} (y_i - y_i')^2$$

Se utilizan técnicas de optimización numérica para minimizar la expresión anterior (algoritmos de mínimos cuadrados, descenso del gradiente, retropropagación de errores, algoritmos Quasi-Newton, etc.).

Ejemplo: Regresión

Ver ficheros Python asociados: **Codigo14.py**

5.4. Validación de modelos

Cuando se experimenta con diferentes modelos, es necesario comparar entre ellos para finalmente seleccionar el más apropiado. Para ello se hace uso de herramientas estadísticas y, en particular, tests de hipótesis. La metodología para validar modelos es la siguiente:

1. Seleccionar uno o varios modelos, y optimizar sus hiperparámetros.
2. Seleccionar un criterio de evaluación (test/train, k-fold…)
3. Seleccionar una métrica para valoración del modelo

4. Para cada modelo:
   - Entrenar el modelo un número dado de veces con valores de parámetros iniciales diferentes y aleatorios cada vez.
   - Obtener los datos de métricas de valoración en train/test, cada vez. Así obtenemos una población de datos de métricas (muestra estadística).

5. Pasar un test de normalidad a todas las muestras estadísticas (Shapiro-Wilk, Jarque-Bera…).

6. Si todas pasaron test de normalidad:
   - Aplicar tests paramétricos a cada par de muestras, ordenadas de mejor a peor muestra (t-Student…).

7. En caso contrario:
   - Aplicar tests no paramétricos a cada par de muestras, ordenadas de mejor a peor muestra (Wilcoxon, Kruskal-Wallis…).

El siguiente ejemplo sirve para valorar cuál modelo es mejor entre un modelo lineal de regresión y un MLP para regresión.

Ejemplo: Validación

Ver ficheros Python asociados: **Codigo15.py**

6. Aprendizaje No Supervisado

El aprendizaje no supervisado es otra modalidad de aprendizaje dentro del aprendizaje automático. Mientras que en el aprendizaje supervisado los patrones tienen una estructura (entrada, salida), y el objetivo es que el modelo de ML aprenda a proporcionar las salidas deseadas a partir de las entradas, en el aprendizaje no supervisado los patrones sólo son de entrada. Por tanto, el objetivo de este tipo de aprendizaje se orienta a descubrir posibles relaciones existentes entre los datos, y conseguir un modelo abstracto de los mismos que permita explicar mediante reglas sencillas cuáles son las relaciones a alto nivel entre los datos. Ejemplos de aprendizaje no supervisado son la extracción de reglas de asociación, el agrupamiento o los mapas auto-organizativos.

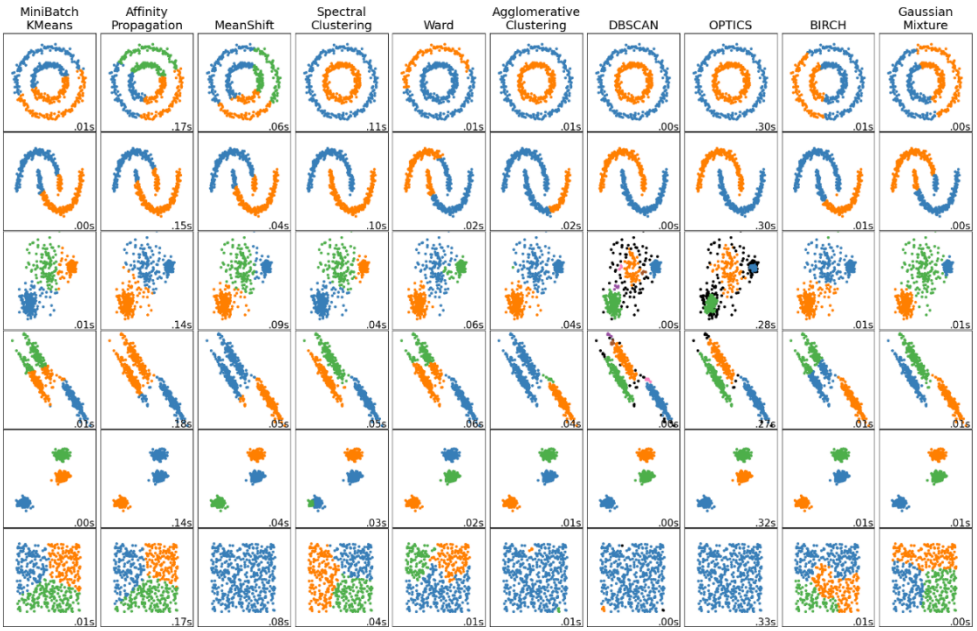
Las reglas de asociación son un tipo de reglas que permiten predecir la ocurrencia de un ítem basándonos en la ocurrencia de otros. Por ejemplo, una regla de tipo "Todo aquel que compra leche también compra pan". Son muy usadas en áreas como la identificación de perfiles de clientes, detección de fraudes, análisis de redes sociales, etc.

Por otra parte, el agrupamiento (clustering) consiste en encontrar reglas que permitan a los patrones agruparse en categorías más abstractas. Por ejemplo, a partir de todos los estudiantes de un curso, encontrar diferentes perfiles existentes de forma automática (que posteriormente podrían tener un significado, como por ejemplo buenos, malos o regulares, etc.).

Finalmente, los mapas auto-organizativos permiten también realizar agrupamientos, mediante la auto-organización de los elementos del conjunto de datos formando categorías más abstractas. Tanto el clustering como los mapas auto-organizativos suelen ir muy ligados a posteriores tareas de clasificación (segmentación de clientes y asignación a tarifa más recomendada, control de publicidad personalizada en redes sociales, etc.).

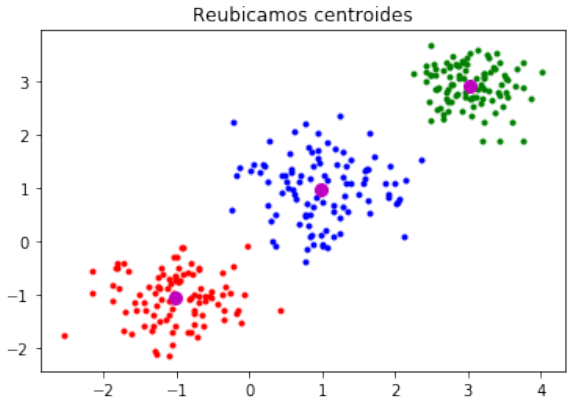
6.1. Técnicas de Agrupamiento

Sklearn proporciona una gran cantidad de técnicas de agrupamiento (clustering):

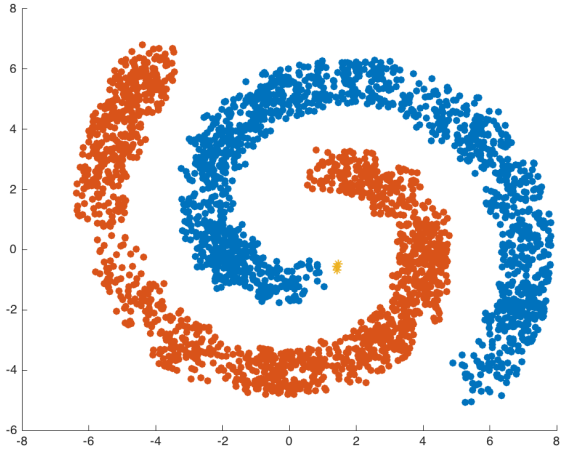


En términos generales, podemos distinguir 3 tipos de técnicas dentro del clustering:

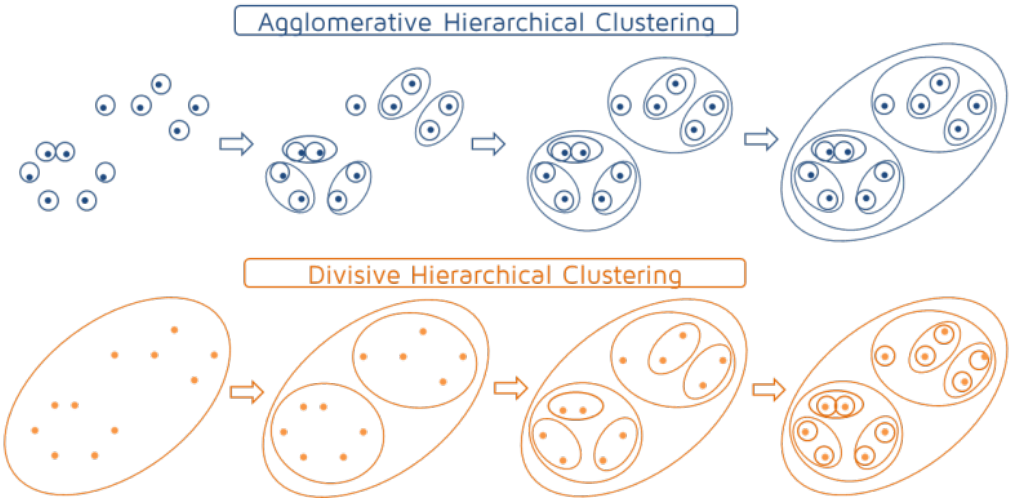
- **Modelos basados en centroides:** Son algoritmos iterativos que, a partir de un número de categorías prefijado, permiten obtener un patrón genérico o plantilla (centroide) que representa a todos los datos relacionados entre sí.



Modelos basados en densidad: Son algoritmos iterativos, que crean un número de agrupamientos no conocido a priori, basándose en cómo de densas son las regiones del espacio.



Modelos basados en conectividad: Se asocia la similitud entre dos patrones de datos basándose en criterios de cercanía. Las técnicas dentro de esta categoría engloban al **clustering herárquico**, que permite, a partir de los datos, crear categorías más abstractas que se van generalizando en un árbol de clusters.



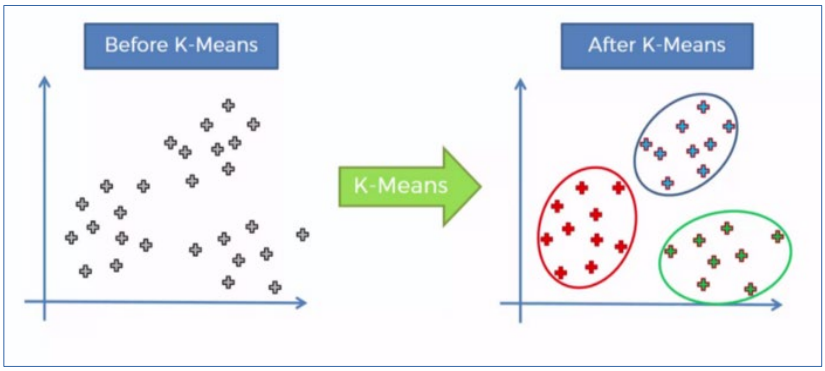
Algunas de las métricas más usadas para poder valorar un método de clustering son:

- **Homogeneidad (H):** Número de elementos de un cluster que son miembros de una misma clase (cuando se dispone de datos etiquetados).
- **Completitud (C):** Número de elementos de una clase conocida que pertenecen al mismo cluster (cuando se dispone de datos etiquetados).
- **V-Score:** cuando los datos están etiquetados, la V-score se calcula como la media armónica entre la homogeneidad y la completitud; es decir:

$$Vscore = \frac{(1+\beta) * H * C}{\beta * H * C}$$

  Donde β suele tener un valor típico igual a 1.
- **R-Score ajustada**, que calcula la similitud entre clusters. Tiene valores bajos si los clusters son independientes (cercanos a 0) y valores cercanos a 1 si son muy similares.

Entre todos los algositmos de clustering estudiados, quizá el más conocido (por su eficiencia, simplicidad y buen comportamiento en general) es el algoritmo **k-Means.** *K-Means* es uno de los algoritmos más conocidos para agregar datos en un número fijado de K clusters. Es un algoritmo iterativo sobre elementos representativos de cada cluster (***centroides***), atendiendo a una medida de distancia entre elementos (la ***distancia euclídea*** suele ser una medida muy usada para comparar distancias entre elementos).



El algoritmo ***K-Means*** sigue el principio de *Expectation-Maximization*. En cada iteración, el paso *Expectation* asigna cada elemento del conjunto de datos al cluster más cercano. A continuación, el paso *Maximization* recalcula el centroide de cada cluster. La **función de pérdida** a optimizar (minimizar) es:

$$J = \sum \sum w_{i,k} \|x^i - c^k\|^2$$

Donde:

- **N** es el número de patrones en el conjunto de datos
- **K** es el número de clusters (establecido a priori)
- $\mathbf{x^i}$ es un patrón dentro del conjunto de datos $\mathbf{x^i=(x^i_1, x^i_2, ..., x^i_n)}$
- $\mathbf{c^k}$ es el centroide del cluster ***k*** (vector $\mathbf{c^k=(c^k_1, c^k_2, ..., c^k_n)}$
- $\mathbf{w_{i,k}}$ tiene valor 1 si el patrón $\mathbf{x^i}$ pertenece al cluster **k**, o 0 en otro caso.

Con estas consideraciones, el algoritmo **K-Means** es el siguiente:

Algorithm C= Kmeans(X={x1, x2, ..., xN}, K, distance) :

1. **C= {c^1, c^2, ..., c^K}** # sample K random data instances from X without repetition
2. **t**= 0, $\mathbf{w_{i,k}(t)}$=0 for all possible **i,k**
3. **Change**= True
4. While **Change**
    1) **t**= **t**+1
    2) For all **i=1..N** : # Expectation step
        1) Initialize $\mathbf{w_{i,k}(t)}$= 0, $\mathbf{N_k}$=0 for all possible **i,k**
        2) Selec **k**= cluster with minimum **distance($\mathbf{x^i}$, $\mathbf{c^k}$)**
        3) Set $\mathbf{s_i(t)=k}$; $\mathbf{N_k= N_k}$+1
    3) Set **Change**=True if there exists **i :** $\mathbf{s_i(t) <> s_i(t-1)}$; or **Change**= False otherwise
    4) For all **k=1..K** : # Maximization step
        1) Calculate $c^k = \sum_{x^j \in Cluster\ k} x^j$
5. Return **C**

El siguiente ejemplo muestra cómo usar **sklearn** para realizar clustering mediante los algoritmos más representativos (**K-Means y DBSCAN**).

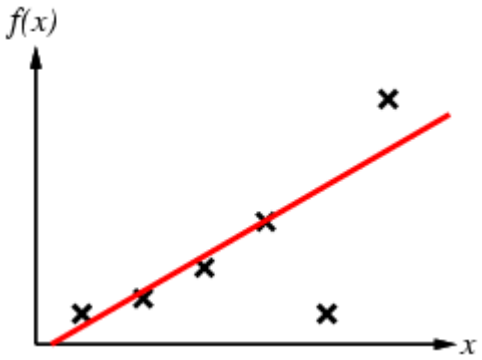
Ejemplo: Clustering
Ver ficheros Python asociados: **Codigo16.py, Codigo17.py**

7. Técnicas de evaluación

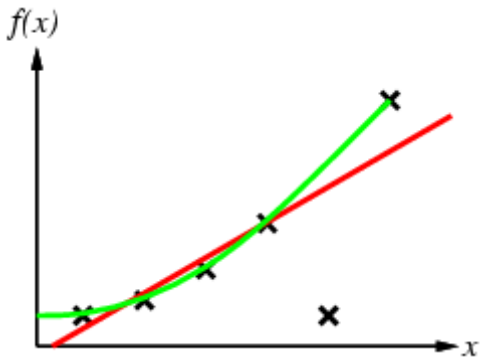
Las técnicas de evaluación de modelos de ML se orientan a la comprobación de que el agente ha aprendido correctamente la tarea que se le ha encomendado. Para ello, es necesario no sólo seleccionar criterios adecuados de calificación y valoración, sino también de la profundidad de lo aprendido. Debido a que el aprendizaje se lleva a cabo principalmente a través de ejemplos (**patrones**), es necesario introducir mecanismos que aseguren que el sistema inteligente ha sido capaz de **abstraer el problema, "comprendiéndolo"** y sin llegar a un tipo de **aprendizaje memorístico** que, ante una situación inesperada, produzca reacciones del agente inesperadas.

Al problema del **aprendizaje memorístico** se le conoce también como **sobre-entrenamiento**, dado que suele suceder cuando el agente ha recibido los mismos estímulos de aprendizaje demasiadas veces y, en lugar de generalizar y aprender las propiedades abstractas del problema, se dedica a ***memorizar*** los patrones de datos para "*mejorar su calificación en la evaluación del modelo*".

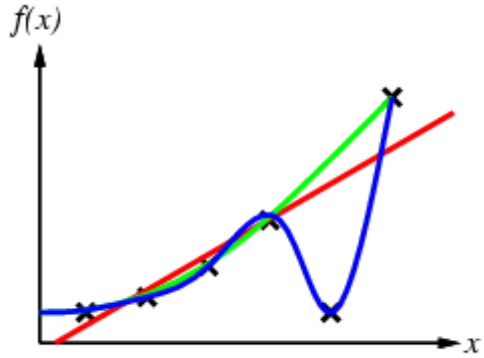
A modo de ejemplo, supongamos que tenemos un problema cuyos patrones E/S son pares de números (x, y). El objetivo del aprendizaje es encontrar una función f(x) tal que f(x)=y (x=entrada, y=salida). Podríamos hacer una hipótesis de modelo lineal; es decir, y=a*x+b, y aprender los valores a y b que mejor ajustan el modelo:



En este caso, parece ser que la hipótesis, aunque parece buen aproximador para los datos con valor de x pequeño, podría no serlo para valores de x más grandes. En su lugar, podríamos realizar una hipótesis de modelo cuadrático, o $y=a^*x^2 +b^*x + c$ y aprender los valores a,b,c (parámetros de nuestro modelo). Daría como resultado una aproximación que parecería mejor:

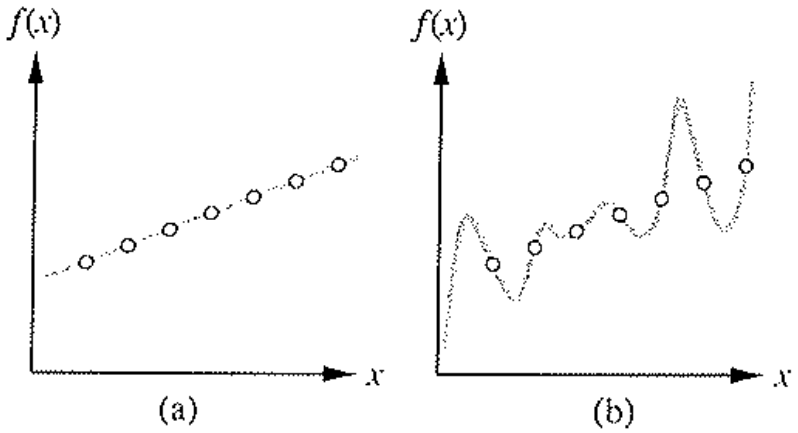


Por último, y para tratar de aproximar aún mejor los datos, podríamos suponer un modelo de tipo spline que pase por todos los puntos del conjunto de datos:



Sin embargo, aunque este modelo ajuste muy bien los datos dados para el aprendizaje, podría comportarse de forma errática ante la presentación de nuevos datos previamente no vistos por el sistema.

Para prevenir la situación anterior, es habitual dividir el conjunto de datos en dos: Uno usado para entrenar el modelo (**training set**) y otro para validarlo (**test dataset** o **validation dataset**). El conjunto de test o validación nunca debe ser presentado al agente para aprender, sino que se usará sólo para comprobar su capacidad de generalización y adaptación a situaciones previamente no estudiadas en el aprendizaje. Un buen agente deberá tener una buena capacidad de actuación tanto en el conjunto de entrenamiento como en el de test y/o validación. En caso de no disponer de suficientes datos, o ser imposible barajar un conjunto de test/validación, primará el criterio de la **navaja de Ockham**; es decir, la elección del modelo más simple que sea consistente con los datos (ejemplo: subfigura (a) de la siguiente imagen):

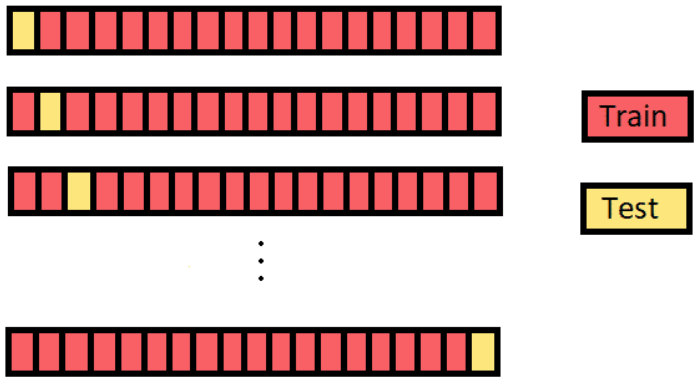


De los párrafos anteriores, podemos concluir que las técnicas de validación, así como los criterios a usar para valorar la idoneidad de un modelo de ML, juegan un papel crucial en el proceso de aprendizaje. **Los criterios de valoración** pueden ser muy variados, y van normalmente ligados al problema a resolver (minimizar los residuos en problemas de aproximación funcional, maximizar la precisión (accuracy) del porcentaje de aciertos en un problema de clasificación, etc.). No obstante, existen un conjunto de **técnicas de validación** comunes a diversos modelos y tipos de problemas, entre las que encontramos como más conocidas las siguientes:

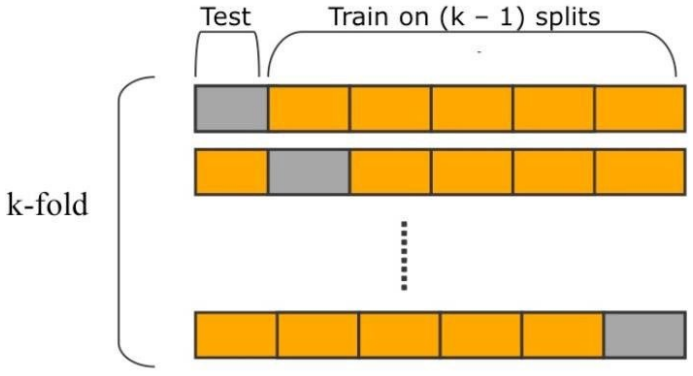
- **Validación simple (también conocida como validación por cruce aleatoria):** Consiste en distribuir todos los patrones de los datos del problema, de forma uniforme pero aleatoria, en dos subconjuntos (training y test). Habitualmente se usa un porcentaje 50%-50% o 70%-30%, aunque no hay una regla fija establecida para ello.

- **Validación por cruce (cross-validation, CV):** En este caso tenemos varias alternativas:

  - **Leave-one-out CV:** Es un proceso iterativo. En cada iteración, se entrena el modelo con todos los patrones del conjunto de datos salvo una, que se deja para validación.



  - **K-fold CV**: Se divide el conjunto de datos en **k** subconjuntos aleatorios pero uniformes. Se entrena un total de **k** veces, con **k-1** subconjuntos y dejando el último subconjunto para test. La idoneidad del modelo se calcula como la media de todos los tests a lo largo de los **k** aprendizajes realizados.



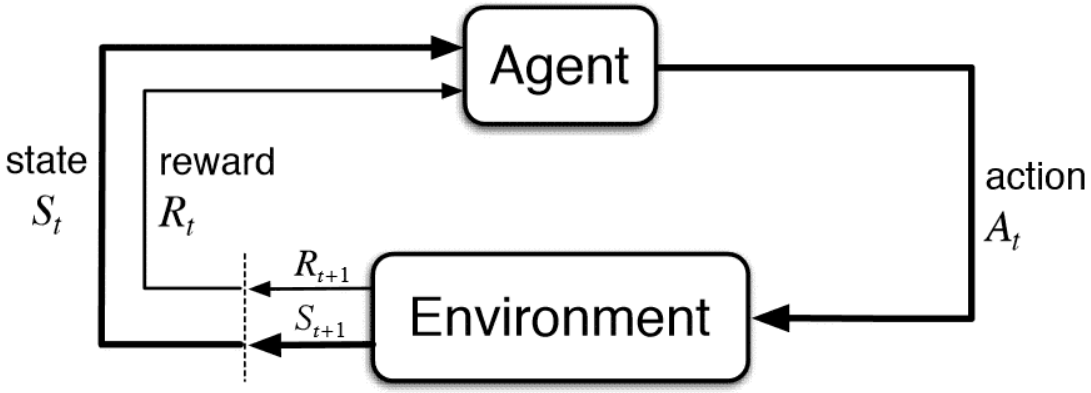
En los siguientes apartados estudiaremos este tipo de técnicas de validación de forma práctica.

Ejemplo: Validación por cruce

Ver ficheros Python asociados: **Codigo18.py**

8. Aprendizaje por refuerzo

El aprendizaje por refuerzo es una rama dentro del aprendizaje automático que, a diferencia de los aprendizajes supervisado y no supervisado, no dispone de un conjunto de datos previo a aprender. En su lugar, un agente que aprende por refuerzo lo hace a partir de su propia experiencia con un entorno, que a priori es desconocido. Los elementos del aprendizaje por refuerzo son los siguientes:



- **Agente:** Entidad capaz de percibir un entorno y realizar acciones sobre él.

- **Entorno:** *"caja negra"*. Contiene un estado. Recibe acciones del agente que afectan (cambian) el estado (el agente no sabe cómo). Devuelve al agente **una observación de** su nuevo estado y una recompensa asociada a la última acción.

El funcionamiento estándar de un sistema de aprendizaje por refuerzo es el siguiente:

- El entorno se encuentra en un estado **s(t),** en un instante de tiempo **t**.
- El agente conoce una observación de **s(t)** y selecciona y ejecuta una acción **a(t)**.
- El entorno cambia su estado de **s(t)** a **s(t+1)** dependiendo de **a(t)**.

- El entorno devuelve una observación de **s(t+1)** y una recompensa **r(t)** al agente.
- Vuelta a empezar, pero en el instante de tiempo siguiente **t=t+1**.

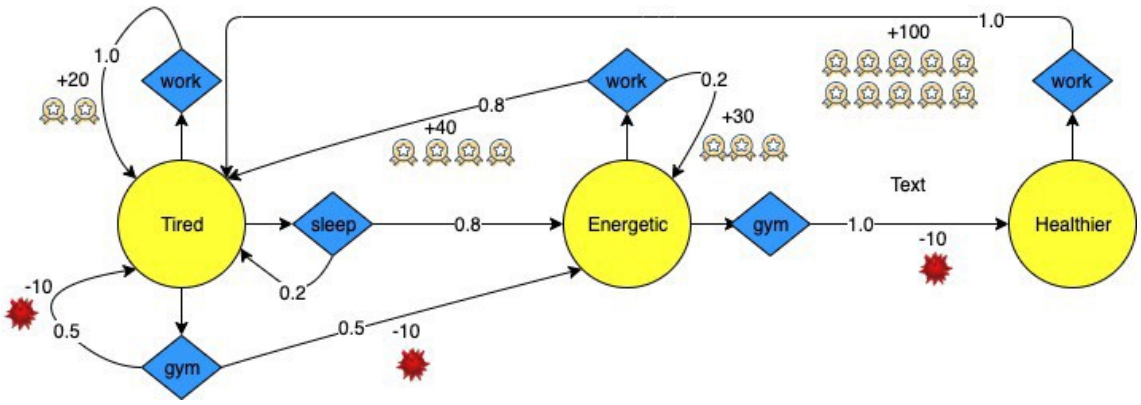
Para que el agente pueda aprender a "***desenvolverse***" bien en el entorno, su objetivo es aprender, para cada estado **s(t),** cuáles son las acciones **a(t)** que mejor puede ejecutar; es decir, **las acciones que más recompensa pueden darle**. No obstante, para evitar aprender acciones que den recompensas inmediatas y que el agente no aprenda el comportamiento correcto a largo plazo, el aprendizaje por refuerzo trata de **maximizar la recompensa total obtenida o "*return*", R**.

$$R = \sum r(t)$$

En este contexto, los elementos que persisten en cualquier sistema de aprendizaje por refuerzo son los siguientes:

- Los **estados s** del entorno (percibidos por el agente como **observaciones**). El estado **s** pertenece a **S** (espacio de estados). Si el entorno se encuentra en un estado **s** y el agente aplica una acción **a**, hay una probabilidad de que se termine en un estado **s' (probabilidad de transición)** : **p(s'|s,a)**
- Las posibles **acciones a** del agente. La acción **a** pertenece a **A** (espacio de acciones, que puede ser discreto y finito, o continuo)
- La **recompensa** de aplicar **a** al entorno, estando en el estado **s** y pasando al estado **s'**: **r(s, a, s') abreviado r(t), valor escalar**. El entorno puede devolver una recompensa **r** dependiendo del estado inicial **s**, la acción tomada **a**, y el estado final al que se llega **s'**.
- **Política (π)**: La forma en que el agente **evalúa sus opciones** y **selecciona** una acción **a** sabiendo que el entorno se encuentra en un estado **s**. Esta política puede ser **determinista** (**a= π(s)**) o **no determinista** (**a ~ π(s)**, también expresado como **π(a|s)**, en cuyo caso leemos *"probabilidad de escoger la acción **a** en el estado **s** con la política **π**"*)

Para poder realizar un modelo matemático de estas componentes, utilizamos los denominados **Procesos de Decisión de Markov, MDP**. Un **MDP** se define como la tupla **(S, A, P, r),** que incluye a **S** (conjunto de estados posibles), **A** (conjunto de acciones posibles), **P** (probabilidades de transición entre estados, p(s'|s,a)), y **r** (función de recompensa **r(s, a, s')**. Por ejemplo, el siguiente diagrama muestra un MDP:



Como se ha comentado, el agente aprende por interacción con el entorno en sistemas de aprendizaje por refuerzo. Esto se realiza a través de la **ejecución de episodios**. Un **episodio o una trayectoria τ** es una secuencia de interacciones entre el agente y el entorno, o **experiencias**, desde el estado inicial s(0) hasta el estado final s(T); es decir:

s(0), a(0), r(0), s(1), a(1), r(1), s(2), a(2), r(2), … , s(T-1), a(T-1), r(T-1), s(T)

Cada **experiencia** conlleva:

- Estar en un estado y conocerlo,
- la selección de una acción en el estado,
- su ejecución, y
- la percepción de la recompensa asociada y el siguiente estado del entorno.

La recompensa total obtenida en una trayectoria la notaremos como **R(τ)**. Para el cálculo de **R(τ)**, y a fin de dar una mayor o menor importancia a acciones inmediatas o futuras, se introduce el denominado **factor de descuento γ**, dentro del cálculo del ***return***, como sigue:

$$R(\tau) = \sum_t \gamma^t r(t)$$

Valores de **γ** cercanos a 1 dan importancia a todas las recompensas obtenidas en la trayectoria, mientras que valores cercanos a 0 dan más importancia a las recompensas inmediatas.

Funciones de valor y ecuación de Bellman

En un sistema de aprendizaje por refuerzo, es razonable pensar en que existirán estados ***"buenos"*** (aquellos que nos permitan alcanzar una buena recompensa total si, a partir de ellos, realizamos una trayectoria con acciones apropiadas) y estados ***"menos buenos"***. Esto nos lleva a definir formalmente la bondad de un estado o **función V(s)** , que indica el valor del estado siguiendo una política **π** dada:

$$V^{\pi}(s) = E_{\tau\sim\pi}[R(\tau)|s_0 = s] = \sum_i R(\tau_i)\pi(a_i|s)$$

Es decir, el ***return*** promedio que se puede obtener desde el estado **s** considerando todas las posibles acciones $\mathbf{a_i}$ que pueden ejecutarse en el estado.

En este caso, para optimizar el comportamiento del agente, queremos encontrar la política óptima $\boldsymbol{\pi}^*$ que nos permita maximizar este valor:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

Otro aspecto en el que podemos estar interesados es en conocer cómo de buena es una acción a. Sin embargo, dicha acción puede ser mejor o peor según el estado en el que se encuentre el entorno, por lo que definimos la bondad del **par estado-acción**, o **función de valor Q** como:

$$Q(s,a) = E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] = \sum_i R(\tau_i)p(s_i|s,a)$$

En este caso, también queremos encontrar cuál es la acción que nos devolverá un valor máximo de ls función Q:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

Tras ejecutar un episodio, se puede calcular el valor de cada estado para ese episodio como sigue (por la propia definición de valor de ***return***):

$$V^\pi(s) = r(s,a,s') + \gamma V^\pi(s')$$

También para la función Q:

$$Q^\pi(s,a) = r(s,a,s') + \gamma Q^\pi(s',a')$$

En términos generales, como pasar a un estado s' depende de una probabilidad p(s'|s,a), los valores de las funciones **V y Q se pueden generalizar (ecuaciones de Bellman)**:

$$\begin{aligned} V^\pi(s) &= E_{a\sim\pi,s'\sim P}[r(s,a,s') + V^\pi(s')] \\ &= \sum_a \pi(a|s) \sum_{s\prime} p(s'|s,a)(r(s,a,s') + \gamma V^\pi(s') \end{aligned}$$

$$Q^\pi(s,a) = \sum_{s\prime} p(s'|s,a)\left[r(s,a,s') + \gamma \sum_{a\prime} \pi(a'|s')Q^\pi(s',a')\right]$$

Mediante un algoritmo iterativo de Programación Dinámica, se puede encontrar ambos valores de forma analítica **si se tiene conocimiento pleno del MDP del problema**:

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma V^*(s')$$

$$Q^*(s,a) = \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right]$$

Algoritmo de Value Iteration

El algoritmo de Value Iteration es un método basado en programación dinámica que, teniendo conocimiento pleno del MDP que modela el entorno (lo cual no es común), nos devuelve la función de valor óptima. Con dicha función sería fácil encontrar cuál es la acción que, en cada estado, debemos escoger para llegar a otro estado que permita maximizar la recompensa total. El algoritmo es como sigue:

1. Calcular una aproximación de $\mathbf{V^*(s)=\max_a \{Q^*(s,a)\}}$, iterativamente.
2. Extraer la política óptima determinista desde la tabla **Q(s,a)**.

```python
def ValueIteration(env, gamma, iteraciones=20, umbralConvergencia=1e-20):

    # Tabla (estado, V(estado)). Inicializada a 0
    Vtable= np.zeros(env.nEstados)

    fin= False
    converge= False
    it= 0
    while not fin: # Ejecutamos hasta criterio de parada
        auxVtable= Vtable.copy()

        # Pasamos por cada estado calculando la tabla Q(s,a)
        for s in range(env.nEstados):
            Qtable= np.zeros(env.nAcciones)
            for a in env.accionesDisponibles(s):
                transP= env.transitionProbs(s, a) # Probabilidades de transición
                for sp,prob in enumerate(transP):
                    if prob>0.0:
                        Qtable[a]+= prob*(env.rewardValue(s,a,sp) + gamma*auxVtable[sp] )

            # Actualizamos Vtable
            Vtable[s]= np.max(Qtable)

        # Pasamos de iteración
        it+= 1

        # Comprobamos convergencia
        converge= np.max(np.fabs(Vtable-auxVtable)) <= umbralConvergencia

        # Criterio de parada
        fin= (it>=iteraciones or converge)

    # Devolvemos la tabla de valores, iteraciones realizadas y si ha convergido
    return Vtable, it, converge
```

```python
def ExtractPolicy(env, Vtable, gamma):
    policy= np.zeros(len(Vtable)) # Creamos la política determinista para cada estado

    for s in range(len(Vtable)):
        Qtable= np.zeros(env.nAcciones)
        for a in env.accionesDisponibles(s):
            transP= env.transitionProbs(s, a) # Probabilidades de transición
            for sp,prob in enumerate(transP):
                if prob>0.0:
                    Qtable[a]+= prob*(env.rewardValue(s,a,sp) + gamma*Vtable[sp] )

        # Actualizamos política
        policy[s]= np.argmax(Qtable)
    return policy
```

A continuación, mostramos una posible implementación del algoritmo para el entorno del MDP de ejemplo:

Ejemplo: Algoritmo de Value Iteration
Ver ficheros Python asociados: **Codigo19.py,**

Entornos desconocidos y estrategias. Algoritmo Q-Learning

El algoritmo de iteración por valor es óptimo y permite encontrar la mejor política si se conoce el MDP del entorno. No obstante, esto no es una situación realista, y el conocimiento que tenemos del mismo es limitado (o incluso puede variar con el tiempo). Ejemplos de estos casos son: Un robot que aprende a andar por sí solo, un agente que realiza trading algorítmico, el control del una planta de tratamiento de aguas residuales, etc. En estos casos, se supone que existe un MDP subyacente, pero que se desconoce (no sabemos p(s'|s,a), r(s,a,s'), etc.).

Existen múltiples técnicas que permiten resolver este tipo de situaciones, y tradicionalmente se engloban dentro de dos categorías:

- Algoritmos de diferencias temporales: Tratan de aproximar la función Q o la función V mediante un proceso iterativo.
- Algoritmos de *policy gradient*: Tratan de optimizar directamente la política, no una función colateral como podría ser la función Q.

Estos métodos han tenido una especial atención en los últimos años, y han surgido numerosas alternativas que mezclan Deep Learning con aprendizaje por refuerzo, dando lugar a algoritmos como Deep Q-Learning, Advantage Actor Critic u otros, capaces de aprender a jugar a videojuegos por sí mismos e incluso a ganar a juegos como el ajedrez a campeones mundiales.

En esta sección introduciremos el algoritmo **Q-Learning**, por ser uno de los más conocidos. **Q-Learning** trata de aproximar el valor de la función **Q(s,a)** de forma iterativa, calculando la diferencia entre el valor de **Q** actual y el valor de **Q** que se obtiene tras recibir la recompensa:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

El valor **α** es una **tasa de aprendizaje**, que permite establecer la velocidad a la que se actualiza la función **Q** de forma iterativa. No obstante, la fórmula anterior tiene un problema: el cálculo de

$$\max_{a'} Q(s',a')$$

Debido a que en un mismo estado sólo podemos ejecutar una acción, y no podemos conocer cuál sería la acción **a'** óptima por repetición de todas las acciones. Además, si la función Q inicial la inicializamos con valores *malos*, la fórmula dejaría de tener validez al siempre escoger una **a'** *mala*.

La solución es incluir **estrategias de exploración** que nos permitan escoger, de forma aleatoria, acciones para *"probar"* otras formas de resolver el problema a las ya conocidas por el agente. En particular, aunque hay múltiples estrategias de exploración, una de las más conocidas es la denominada como ε-greedy, que actúa como sigue:

1. Obtener un valor aleatorio **p** en U(0,1)
2. Si **p<** ε, entonces escoger una acción a al azar.
3. En otro caso, seleccionar la mejor acción conocida hasta el momento a que haga $\max_a$ Q(s,a).

El valor ε es un hiperparámetro del algoritmo. Normalmente, se suele ir modificando con las iteraciones, permitiendo un valor ε elevado al comienzo del aprendizaje (explorar muchas acciones) y un valor reducido al final (potenciar el conocimiento aprendido):

```python
# Función que devuelve una acción en 0..NumActions-1 según la política e-greedy,
#  para el estado S, teniendo política de agente pi(s) determinista
#  dada en "politicaAgente"
#  Como entrada también se tienen los valores aproximados Q(s,a) en Q, y epsilon
def politicaEpsilonGreedy(S, NumActions, politicaAgente, Q, epsilon):

    if np.random.rand() < epsilon: # Politica aleatoria uniforme
        return np.random.randint(NumActions)

    else:
        return politicaAgente(S, Q)
```

```python
def politicaAgente(S, Q):
    return np.argmax(Q[S, :])
```

Con estas consideraciones, el algoritmo Q-Learning es el siguiente:

1. **(Parámetros)** Inicializar ε_0, ε_f, ε_{epi}, $\varepsilon=\varepsilon_0$
2. **(Inicializar)** Generar función Q inicial
3. **(Generar N episodios)**. Desde **ep**=1 hasta N, hacer:
    1) **(Reset del entorno) s**= estado inicial
    2) **(Bucle Q-Learning)** Para cada experiencia en el episodio:
        1) Selección de acción **a** con política e-greedy en estado **s**
        2) Ejecutar acción **a** y obtener estado **s'** y reward **r**
        3) Aplicar regla de actualización Q-Learning
        4) Moverse al siguiente paso, **s= s'**
    3) **(actualizar exploración)** Actualizar valor de ε como:

$$\epsilon = \max(\epsilon_f, e_0 ep(\epsilon_f - \epsilon_0)/\epsilon_{epi}$$

El siguiente código muestra cómo usar Q-Learning para resolver el MDP de ejemplo:

Ejemplo: Algoritmo de Q-Learning

Ver ficheros Python asociados: **Codigo20.py,**

Software para aprendizaje por refuerzo

En los últimos años han surgido numerosas herramientas para realizar aprendizaje por refuerzo, en especial en hibridación con redes neuronales profundas (***Deep Reinforcement Learning***). En el siguiente módulo, dedicado a **Deep Learning**, haremos menciones a ellas con mayor detalle. No obstante, cabe destacar una herramienta para entrenamiento de agentes (no necesariamente con Deep Learning), de la organización OpenAI. La herramienta **gym** permite disponer de múltiples entornos de muchos tipos (entradas y/o salidas continuas, gran cantidad de acciones, espacio de estados grande, etc.). También contempla la integración con motores 3D para simulación, y un conjunto de videojuegos Atari para aprender por refuerzo. Su instalación se realiza por línea de comandos:

pip install gym

para una instalación básica, o

pip install gym[atari]

si también se desea descargar los entornos para Atari. El siguiente código fuente muestra cómo usar el paquete de software para aprender por Q-Learning el entorno de texto denominado Taxi:

Ejemplo: Algoritmo de Q-Learning con Gym
Ver ficheros Python asociados: **Codigo21.py,**

9. Ajuste de hiperparámetros y AutoML

Los **hiperparámetros** son parámetros que se utilizan para ajustar el comportamiento genérico de los modelos, o bien de los algoritmos de aprendizaje. Por ejemplo, el número de clusters inicial en K-Means, el tipo de kernel en SVM, el número de neuronas de una red neuronal artificial, la tasa de aprendizaje de un algoritmo, etc. El correcto o incorrecto valor de estos hiperparámetros puede conducir a que se obtengan malos resultados en la evaluación de un experimento, por lo que resulta crítico encontrar cuáles son los mejores antes de decidirse por un modelo u otro ya entrenados.

Por otra parte, la selección del mejor modelo de Machine Learning que debemos usar para resolver un problema también suele ser una tarea tediosa. Requiere la replicación de bastante código fuente, particularizando para cada modelo a probar uno o varios ficheros. Afortunadamente, esta tarea se ha automatizado durante la última década, surgiendo software especializado en la búsqueda automática de modelos. Este tipo de software recibe el nombre de **Automated Machine Learning**.

Ajuste de hiperparámetros

Tradicionalmente, el proceso de ajustar los hiperparámetros se ha realizado mediante una **prueba de ensayo y error**, donde el científico/usuario realiza diversas pruebas y va cambiando valores de hiperparámetros según los resultados que se obtienen tras un aprendizaje. No obstante, existen varias técnicas automatizadas para el ajuste de hiperparámetros:

- **Random Search:** Se realiza una búsqueda aleatoria sobre diferentes valores de hiperparámetros, para dar con un subconjunto de valores que luego pueden ajustarse con más precisión, bien manualmente o bien por otro procedimiento automatizado.

- **Grid Search:** Se proporciona un rango de valores deseado para probar cada uno de los hiperparámetros que se tienene, y se realiza una búsqueda combinatoria exhaustiva. La ventaja de esta técnica es que se puede llegar a conocer los hiperparámetros óptimos con seguridad; sin embargo, tiene el inconveniente de requerir un gran coste computacional (y tiempo).

Ambas técnicas se incluyen dentro de **sklearn**. El siguiente ejemplo muestra cómo usarlas:

Ejemplo: Ajuste de hiperparámetros
Ver ficheros Python asociados: **Codigo22.py,**

Búsqueda de modelos automatizada

La búsqueda de modelos automatizada, o **Automated Machine Learning (AutoML)**, pretende ayudar en el proceso de la selección del mejor modelo que puede resolver un problema dado. En particular, **sklean** tiene integrada una herramienta a tal fin (**auto-sklearn**), aunque la biblioteca es bastante joven. No obstante, tiene posibilidades de realizar AutoML para problemas de clasificación y de regresión, entre los más comunes. Otras bibliotecas orientadas a tal fin son las siguientes (https://neptune.ai/blog/a-quickstart-guide-to-auto-sklearn-automl-for-machine-learning-practitioners):

Name of AutoML framework	Created by	Documentation/ GitHub	Open-source
Auto-sklearn	Matthias Feurer, et al.	GitHub/ Documentation	Yes
Auto-xgboost	Janek Thomas et al.	GitHub	–
GCP-Tables	Google	Source	No
AutoGluon	Amazon	GitHub/ Source	Yes
AutoML-azure	Microsoft	Source	No
GAMA	Pieter Gijsbers et al.	GitHub/ Source	Yes
Auto-WEKA	Chris Thornton et al.	GitHub/ Documentation	Yes
H2O AutoML	h2o.ai	GitHub	Yes
TPOT	Randal S. Olson, et al.	GitHub/ Documentation	Yes
ML-Plan	Marcel Wever et al.	GitHub/ Documentation	Yes
Hyperopt-sklearn	Brent Komer et al.	GitHub/ Documentation	Yes
SmartML	Mohamed Maher et al.	GitHub	Yes
MLJAR	MLJAR Team	GitHub/ Documentation	Yes

10. Retos de Machine Learning: Introducción a Kaggle

Tradicionalmente, el acceso a datos ha sido el mayor de los problemas de empresas y entornos académicos para validar el funcionamiento de modelos de aprendizaje automático, en muchas áreas de conocimiento. No obstante, eso ha cambiado en la era actual del Big Data, donde fácilmente se pueden encontrar conjuntos de datos (algunos de dimensiones estratosféricas) para realizar tareas de aprendizaje. Existen plataformas en internet dedicadas a la difusión de estos conjuntos de datos (UCI Data Repository, Data.World, Kaggle…). Entre todas, descata la plataforma **Kaggle** dado que no sólo pone a disposición de los usuarios múltiples conjuntos de datos, sino que también proporciona código fuente de ejemplo y formas de validar y competir entre diferentes personas que resuelven el mismo problema y, en ocasiones, incluso se llega a poder recibir recompensas por resolverlo (por ejemplo, empresas que lanzan una competición para resolver un problema).

kaggle

Dentro de la plataforma **Kaggle** encontramos distintos recursos:

- **Competiciones:** Datasets dispuestos para la comunidad Kaggle. Cualquiera puede apuntarse a una competición y enviar su solución.
- **Datasets:** Habitualmente de competiciones ya finalizadas, aunque también encontramos conjuntos de datos no ligados a ninguna competición.

- **Código fuente:** Ejemplos de cómo usar la plataforma, pero también de cómo acceder, leer y preprocesar conjuntos de datos.
- **Foros:** Diversos temas de discusión entre los que destacan el uso de la plataforma, o resolución de algún problema/dataset particular.
- **Recursos de aprendizaje:** Diferentes tutoriales (todos en inglés) para aprender sobre una temática concreta.
- **Ranking de usuarios:** Un ranking sobre los ganadores de diversas competiciones y acceso a recursos de Kaggle.
- **Tutoriales de Kaggle:** Cómo sacar el mayor partido a la plataforma.

vuela

PLATAFORMA
DIGITAL DE
ANDALUCÍA

Junta de Andalucía