Tema 05. Otras técnicas de aprendizaje automático supervisado

Autor: Ismael Sagredo Olivenza

5.1 Otras técnicas de Aprendizaje automático supervisado

En este tema vamos a describir otras técnicas de aprendizaje automático diferentes a las redes de neuronas.

Aunque la redes de neuronas han alcanzado un gran popularidad y son el estándar del Machine Learning actual, es recomendable conocer otras técnicas ya que cuentan con algunas ventajas sobre las redes de neuronas y para ciertos problemas pueden ser suficientemente buenas.

5.1.1 Ventajas e inconvenientes de las redes de neuronas

Ventajas:

- Tolerancia a fallos (ruido de lso datos de entrada)
- Modelos escalables
- Pueden resolver problemas no lineales

Inconvenientes:

- Coste computacional elevado
- Algoritmo de caja negra.

5.2 K-Nearest Neighbor o k vecinos más cercanos

También conocido como **KNN** por sus siglas en inglés.

El algoritmo no intenta extraer una característica de los datos haciendo un proceso de entrenamiento exhaustivo con ellos. Simplemente los almacena para recuperarlos en el futuro.

Cuando se presenta un nuevo caso, éste es comparado con la base de casos y se extrae aquel más similar (o los K más similares) para comprobar cual era la clase a la que pertenecía. Se asume que si el caso es similar, la solución será similar.

Se necesita definir una función de similitud entre dos ejemplos. La calidad de esta función de similitud es crítica para éste algoritmo. Si la función de similitud no es capaz de medir correctamente la similitud que existe entre dos casos dados, la asunción de que a similares casos se espera que la instancia sea de la misma clase pierde vigencia y KNN deja de clasificar correctamente.

La solución a seleccionar depende del K elegido. Si K = 1 será la instancia recuperada más similar. Si K > 1 habrá que tomar una decisión de qué clase elegimos. Normalmente el criterio más lógico es **la clase mayoritaria**. Si la salida es un número real, podríamos hacer la media.

Hay multitud de **medidas de distancia** que podemos usar:

- **Distancia Euclídea**: Para cada atributo de X e Y

$$D - Euclidea(X, Y) = \sqrt[2]{\sum_{i=1}^{N} (x_i - y_i)^2}$$

- **Distancia de Manhattan**: Para cada atributo de X e Y

$$D - Man(X, Y) = \sum_{i=1}^{N} |x_i - y_i|$$

- **Distancia de Minkowski**: generalización de la euclídea o la de manhattan donde el exponente y la raíz pueden ser cualquier número.

$$D - Minkowski(X, Y) = \sqrt[p]{\sum_{i=1}^{N} (x_i - y_i)^p}$$

- **Distancia de edición (Levenshtein distance)**: el número de cambios que hay que hacer para convertir una instancia en otra. Generalización de Hamming.
  - casa → cala (sustitución de 's' por 'l') = 1
  - casa → calle (s por L, a por e, inserción de l) = 3

- **Distancia de Mahalanobis**: la similitud entre dos variables aleatorias multidimensionales. Se diferencia de la distancia euclídea en que tiene en cuenta la correlación entre las variables aleatorias.

Formalmente, la distancia de Mahalanobis entre dos variables aleatorias con la misma distribución de probabilidad $\vec{x}$ e $\vec{y}$ se calcula de la siguiente forma:

$$D - Mahalanobis(\vec{x}, \vec{y}) = \sqrt[2]{(x - \mu)^T * C^-1 * (x - \mu)}$$

Donde μ es la media de los datos y C-1 es la inversa de la matriz de covarianzas.

```python
def mahalanobis(x=None, data=None, cov=None):
    """Computa la covarianza entre cada fila de los datos
    x    : Vector o matriz de datos con p columnas
    data : ndarray el total de datos de la distribución
    cov  : matriz de covarianzas (p x p) de la
    distribución. Si no se invoca se establece como none None
    """
    x_minus_mu = x - np.mean(data) #Cálculo de la media
    if not cov: #Calculamos la covarianza i
        cov = np.cov(data.values.T)
    inv_covmat = sp.linalg.inv(cov) # inversión de la matriz de covarianza
    left_term = np.dot(x_minus_mu, inv_covmat) # producto escalar entre
    #x menos la media y la inversa de la matriz de covarianza
    mahal = np.dot(left_term, x_minus_mu.T)
    # producto escalar entre la traspuesta de x menos la media y el cálculo anterior
    return mahal.diagonal() # devolvemos la diagonal de la matriz
```

Nótese que no calculamos la raiz cuadrada, por lo que en realidad estamos calculando el cuadrado de la distancía de mahalanobis

otra aproximación es ponderar el peso que tiene cada uno de los atributos en el cálculo de la distancia. Esto implica **introducir información del dominio** ya que debemos saber o intuir que ciertas variables aportan mas información a la hora de decidir los pesos.

$$D - Euclidea - Ponderada(X, Y, P) = \sqrt[2]{\sum_{i=1}^{N} p_i (x_i - y_i)^2}$$

Donde P es el vector de pesos asociado a cada entrada.

En principio podemos ponderar cualquier distancia pero en unas tiene más sentido que en otras.

5.2.1 Ventajas de KNN

- No paramétrico (salvo que usemos distancias ponderadas). No hace suposiciones explícitas sobre la forma de los datos.
- Algoritmo simple tanto de explicar como de interpretar.
- Alta precisión (relativa). Es bastante alta, aunque no superior a otros modelos más sofisticados. Pero a pesar de su aparente simpleza, si se elige correctamente la distancia puede ofrecer resultados bastante buenos
- El proceso de entrenamiento es inmediato

5.2.2 Inconvenientes del KNN

- Es muy sensible a los atributos irrelevantes. Hacer una buena selección de atributos relevantes es fundamental.
- Es sensible al ruido, ya que, si un ejemplo es un mal ejemplo de entrenamiento y es el seleccionado como el más similar, daremos una solución errónea. Esto se puede mitigar haciendo que K sea grande ya que reduce el ruido
- La ejecución es lenta si hay muchos datos de entrenamiento, ya que tiene que procesar todos los datos. Existen métodos para optimizarlo usando partición espacial pero aún asi es costoso
- Es caro en memoria ya que ocupa mucha memoria si hay muchos casos (Y tiene difícil solución, salvo limitar la memoria de trabajo)

Ejemplo de KNN en SKLearn

```
print(len(iris.data))
X_train = iris.data[:-20]
y_train = iris.target[:-20]
X_test = iris.data[-20:]
y_test= iris.target[-20:]

from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors = 1)

model.fit(X_train, y_train)

prediction = model.predict(X_test)
print("Predicción")
print(prediction)
print("Test")
print(y_test)

model.score(X_test, y_test)
```

5.3 CBR

Almacenamiento de sabiduría en base a ejemplos. Sólo necesitamos ejemplos y como se resolvieron.

CBR solo necesita:

- Una base de casos ya resueltos
- Una medida de similitud de casos
- Un conocimiento de como adaptar los casos si no sean iguales.

Se diferencia de KNN puro y duro en varios puntos:

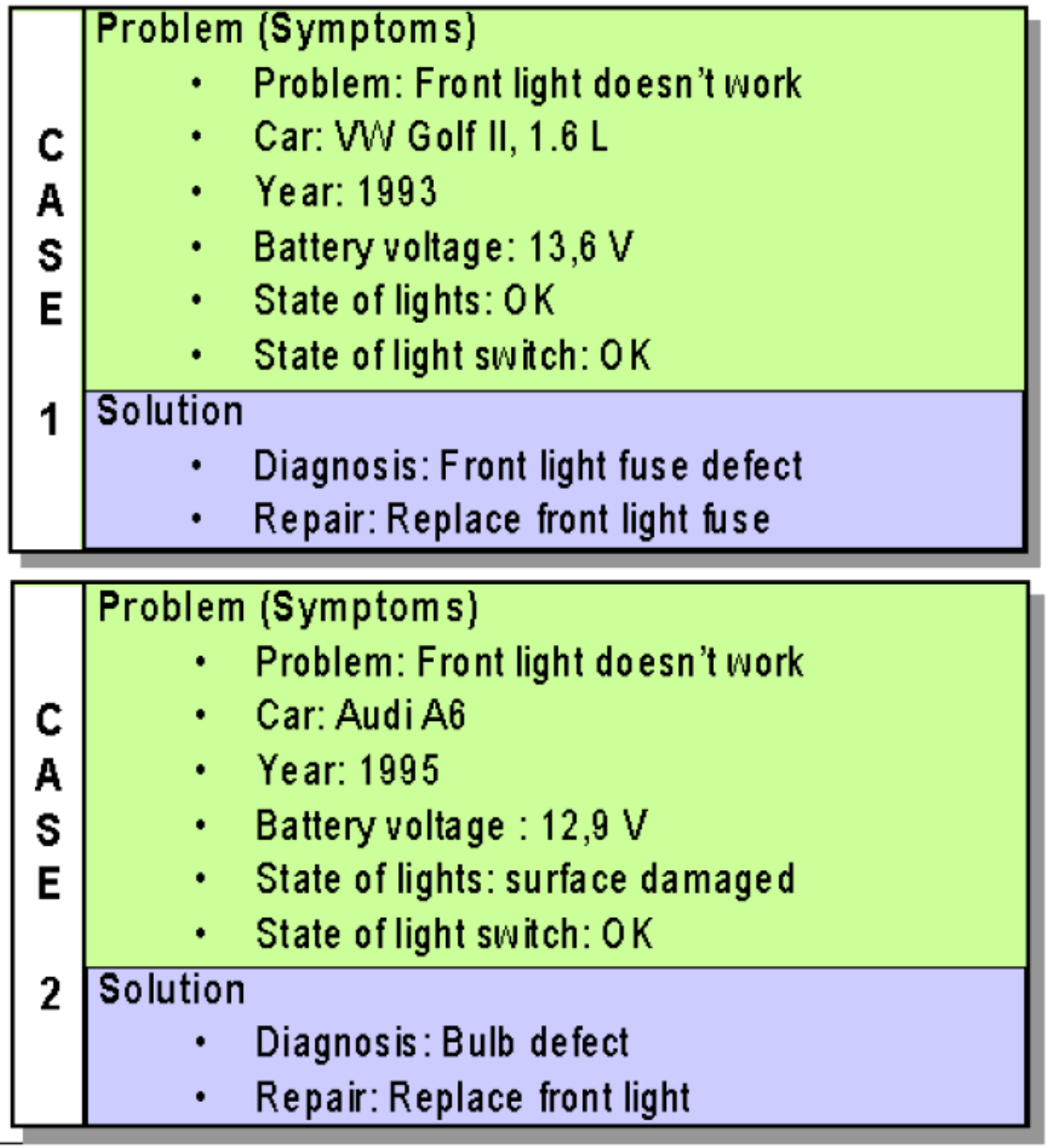
- No sólo se basa en tuplas de ejemplos atributo = valor. La representación puede ser más compleja (ontología, marco, etc)
- Los casos se indexan para mejorar su recuperación
- Las soluciones se adaptan si no es directamente aplicable.
- La base de casos se va actualizando y manteniendo
- Hay interacción con los usuarios (el usuario puede añadir conocimiento extra que ayude al sistema)

The University of Auckland

Each case describes one situation

Cases are independent of each other

Case are not rules



5.3.1 Similitud

Se computa la similitud de un caso con respecto a otro.

Igual que en KNN cada atributo puede tener un peso diferente en la similitud

La similitud en estos casos será una suma ponderada de las similitudes de cada atributo.

Cada atributo puede tener una función de similitud acorde a sus características

5.3.2 Adaptación de las soluciones

Se obtiene un caso muy similar, pero la solución no es aplicable (Por ejemplo la solución no utiliza los mismos parámetros) Ejemplo:

- Tenemos un golpe con el coche y se nos rompe la luz de freno
- El caso más similar es uno en el que se ha roto la luz delantera.
- La solución del caso recuperado es cambiar la luz delantera.

No podemos aplicar directamente la solución porque el caso no es exactamente el mismo y no es aplicable la solución. Hay que adaptarla => Sustituir la luz de freno.

5.3.4 Aprendizaje del nuevo caso

- Si el nuevo diagnostico es correcto, podemos almacenar el caso como caso de ejemplo
- Para ello necesitamos un sistema que nos de feedback de si la nueva solución es correcta.

¿La base de casos puede crecer indefinidamente?

No, si la base de casos aumenta, el tiempo de recuperación de un caso aumenta. Hay que determinar cual es la base de casos máxima permisible los tiempos de respuesta que requiera el sistema. El tamaño en memoria tambien es importante.

5.3.5 Medidas de similitud: TF/IDF

TF/IDF: producto de dos medidas, frecuencia de término y frecuencia inversa de documento. $S(t,d) = tf(t,d) * idf(t,D)$

$$tf(t,d) = \frac{frecuencia(t,d)}{MAX(f(i,d))\forall i \in d}$$

$$idf(t,D) = log(\frac{|D|}{1 + count(t \in D)})$$

$count(t \in D)$ indica el número de documentos donde aparece el termino t. d documento actual, D colección de documentos y t el termino a buscar.

Otra forma de calcular TF/IDF:

El peso del término t en el documento d es igual al número de veces que aparece el término t en el documento d dividido entre la importancia del término t, que se calcula como el número de veces que aparece el término t, dividido entre el número de documentos que hay en el corpus de búsqueda.

$$w(t,d) = \frac{|t \in d|}{imp(t)}$$

$$imp(t) = \frac{|t \in D|}{|D|}$$

- **Similitud del coseno**: dados dos vectores a y b de dimensión N el coseno se calcula como:

$$cosSim(a,b) = \frac{\sum_i a_i, b_i}{\sqrt{\sum_i a_i^2}\sqrt{\sum_i b_i^2}}$$

soft cosine es igual pero en vez de calcular el producto o la suma de de los valores de los atributos normalizados, disponemos de una matriz de similitud calculada para todos los valores de los vectores a y b.

- **Correlación de Pearson**: el coeficiente de correlación de Pearson es una medida de dependencia lineal entre dos variables aleatorias cuantitativas. A diferencia de la covarianza, la correlación de Pearson es independiente de la escala de medida de las variables.

$$\rho(a,b) = \frac{Cov(a,v)}{\sqrt{Var(a)Var(b)}}$$

5.4 Árboles de decisión

Son muy útiles para visualizar las diversas opciones de las que se dispone para resolver un problema o modelar un comportamiento. Se pueden convertir en reglas.

Se pueden aplicar métodos de inducción hacia atrás para descubrir el razonamiento que encierra la lógica del sistema.

Un **árbol de decisión** está compuesto de dos tipos de nodo, un **nodo condicional** y una **clase**. Los nodos condicionales son los nodos internos del árbol y la clase, los nodos hoja o terminales.

Cada condición (nodo condicional) es una pregunta para el sistema acerca de una variable y tiene dos posibilidades, que sea cierta o falsa. En función de eso seguirá un camino y otro. Los nodos finales son nodos hoja y nos dice la clase.

Fuente (https://es.wikipedia.org/wiki/Árbol_de_decisión#/media/Archivo:Arbol_decision.jpg)

Algunas de las implementaciones más famosas de estos algoritmos son el **ID3, J48 o C4.5 (clasificación) y M5 (regresión)**

Este algoritmo utiliza la **entropía** (medida de incertidumbre o de desorden) para ayudar a decidir qué atributo debe ser el siguiente en ser evaluado en el árbol. Es decir, el atributo seleccionado es aquel que deja la información más ordenada o, mejor clasificada.

$$Entropia(s) = \sum_{n=1}^{c} -p_i * Log_2 p_i$$

Donde c son los posibles valores de clasificación, S es el conjunto de todos los ejemplos y p_i es la proporción de ejemplos de S que están en la clase i

Y la ganancia:

$$Ganancia(S, A) = Entropia(S) - \sum_{v \in V(A)}^{|A|} \frac{|sv|}{|s|} Entropia(S_v)$$

Donde V(A) es el conjunto de todos los valores posibles para el atributo A y Sv es un subconjunto de S para el cual el atributo A tiene el valor v

Como selecciona el atributo más prometedor de entre todos, podemos concluir que realiza una **búsqueda voraz** entre los mejores atributos. Después aplica **Divide y vencerás** recursivamente con el problema.

5.4.1 Algoritmo

```
ID3(E,A,N):N
E: ejemplos, A: atributos, N: nodoRaíz
-------------------------------------
Si vacio(A) o Todos lso ejemplos de E pertenecen a una clase
  N.Clase = ClaseMayoritaria(E)
si no
  a = MejorAtributo(A) => aplicando entropía
  A = A - a => quitamos a
  Para cada valor v del atributo a de todas las instancias
    newNode = CrearNodo(a,v)
    N.AddHijos(newNodo)
    E = E - aquellos cuyo valor de de a sea v
    ID3(E,A,newNode)
return N
```

Ejemplo

Sitio de acceso A1	Cantidad gastada A2	Vivienda A3	Última compra A4	Clase
1	0	2	Libro	Bueno
1	0	1	Disco	Malo
1	2	0	Libro	Bueno
0	2	1	Libro	Bueno
1	1	1	Libro	Malo
2	2	1	Libro	Malo

Elegimos el mejor atributo:
$A1 = \frac{1}{6}.I_{10} + \frac{4}{6}.I_{11} + \frac{1}{6}.I_{12}$ Donde I_{1i} son los posibles valores del atributo 1 = {0,1,2}
Ahora calculamos I_{10}, I_{11} e I_{12} usando la Entropía:
I_{10} tenemos para la clase 0 (Bueno) y valor 0 un total de 1 ejemplos. Para la clase 1 (Malo) y valor 0 tenemos 0 ejemplos. El total de elementos con el valor 0 es 1, de ahi podemos deducir lo siguiente:

$I_{10} = -\frac{1}{1}log_2\frac{1}{1} - \frac{0}{1}log_2\frac{0}{1} = -1 * 0 + 0 = 0$ y para el resto...
$I_{11} = -\frac{2}{4}log_2\frac{2}{4} - \frac{2}{4}log_2\frac{2}{4} = 1$ (hay 2 buenos y 2 malos)
$I_{12} = 0$ porque hay 0 buenos y 1 malo

Aplicando la formula tenemos que:

$A1 = \frac{1}{6}.0 + \frac{4}{6}.1 + \frac{1}{6}.0 = 0,66$

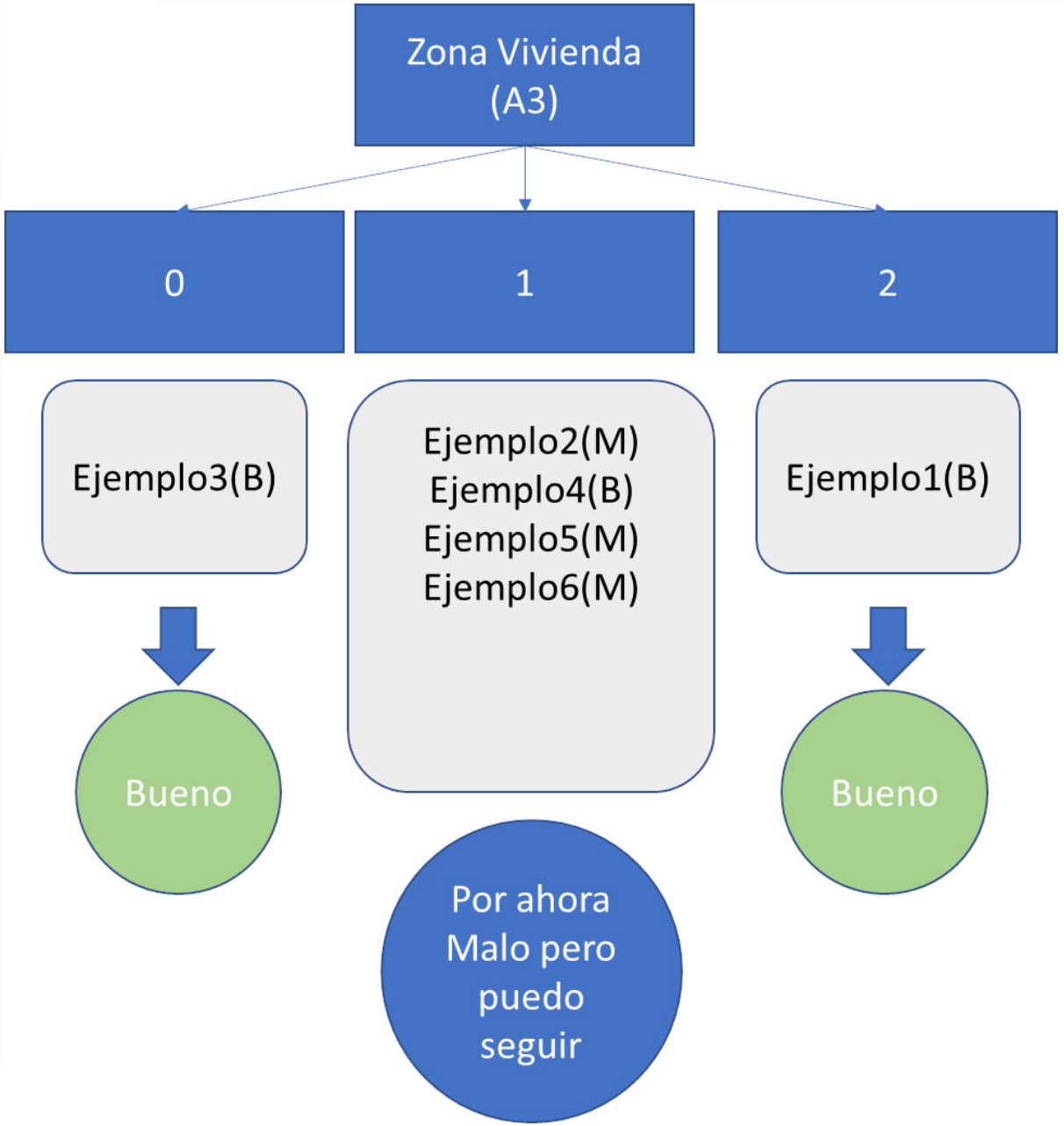
$A2 = 0,79$

$A3 = 0,54$

$A4 = 0,81$

El atributo seleccionado sería el de menor entropía (mayor orden) y por tanto seria el A3.

Esto ya nos deja el árbol con algunos terminales. Pero como puedo seguir podría seguir expandiendo y repitiendo la operación par el resto de atributos (A1,A2,A4)

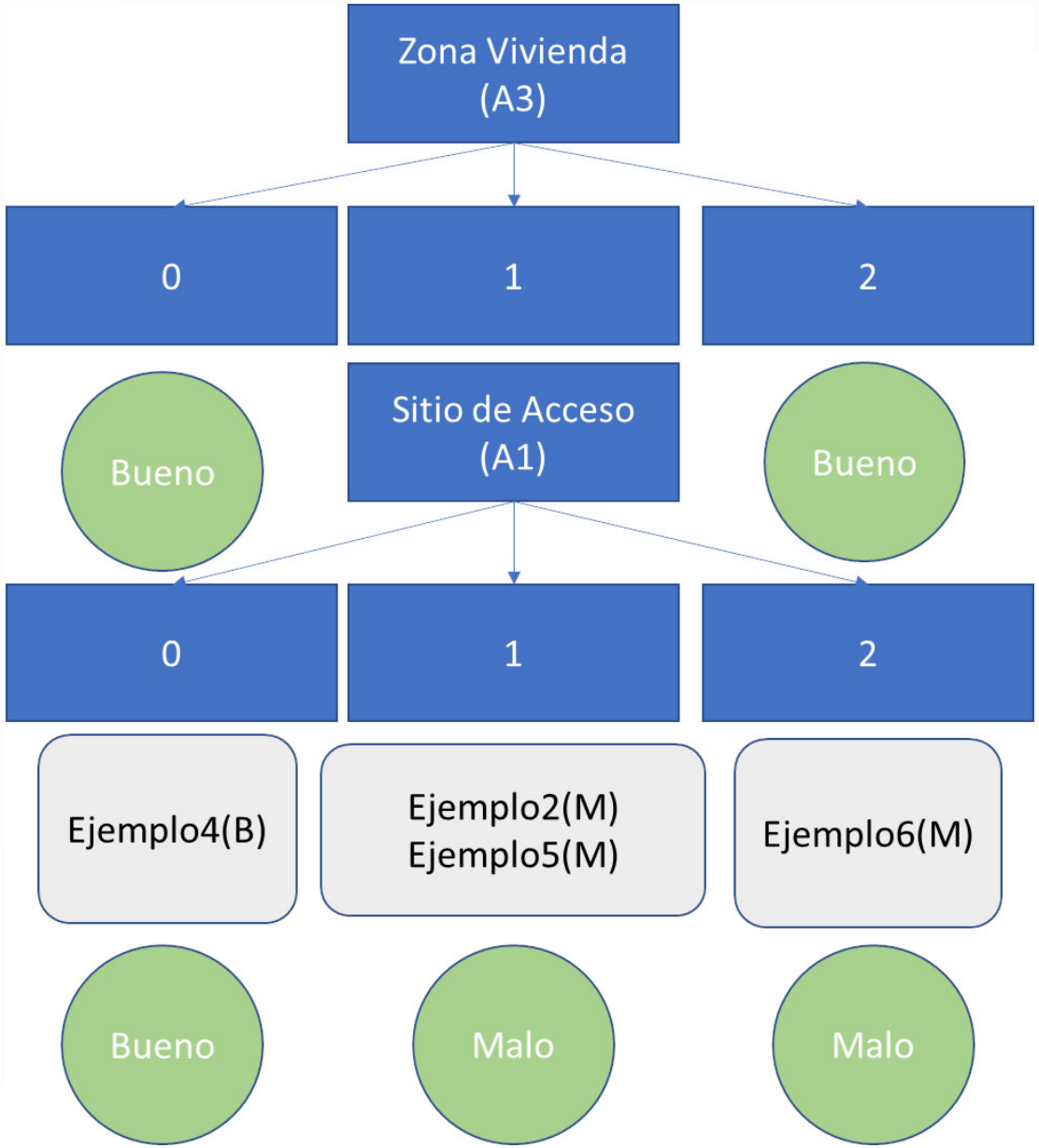

Zona Vivienda
(A3)
0
1
2
Ejemplo3(B)
Ejemplo2(M)
Ejemplo4(B)
Ejemplo5(M)
Ejemplo6(M)
Ejemplo1(B)
Bueno
Bueno
Por ahora
Malo pero
puedo
seguir

El siguiente atributo sería A1 sitio de acceso

$A1 = 0$
$A2 = 0,5$
$A4 = 0,23$

Que nos dejaría el árbol totalmente ordenado.

Nótese lo comentado antes, el DT hace una búsqueda voraz, jamas se replantea si habiendo puesto otro nodo inicialmente la configuración hubiera sido mejor (que en este caso no, pero en otros podría serlo.


Zona Vivienda
(A3)
0
1
2
Bueno
Sitio de Acceso
(A1)
Bueno
0
1
2
Ejemplo4(B)
Ejemplo2(M)
Ejemplo5(M)
Ejemplo6(M)
Bueno
Malo
Malo

5.4.2 Representación gráfica del algoritmo.

Los árboles de decisión dividen el espacio de soluciones mediante hiperplanos fijando una de las variables a un valor frontera.

5.4.3 Ventajas

- El entrenamiento es muy rápido
- Es fácil de interpretar los resultados por un humano, es un algoritmo de caja blanca.
- Para algunos problemas consigue una buena precisión.
- Se pueden convertir fácilmente en reglas.
- No requiere una preparación de los datos demasiado exigente.
- Puede trabajar con variables cualitativas y cuantitativas.

5.4.4 Desventajas

- Es muy dependiente al ruido de la entrada
- Los árbol de decisión tienden al sobre-entrenamiento
- No se puede garantizar que el árbol generado sea el óptimo
- Hay conceptos que no son fácilmente aprendibles pues los árboles de decisión ya que las particiones del espacio de soluciones que puede hacer son aquellas que son representables mediante una sucesión de hiperplanos. Si no hay una aproximación lineal al problema, puede que den un modelo poco efectivo.
- Se recomienda balancear el conjunto de datos antes de entrenar.

Ejemplo de DT en SKLearn

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_text
iris = load_iris()
decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
X_train = iris.data[:-20]
y_train = iris.target[:-20]
X_test = iris.data[-20:]
y_test= iris.target[-20:]

decision_tree = decision_tree.fit(X_train, y_train)
r = export_text(decision_tree, feature_names=iris['feature_names'])
print(r)
decision_tree.score(X_test, y_test)
```

Ejemplo de DT en SKLearn

```
|--- petal width (cm) <= 0.80
|    |--- class: 0
|--- petal width (cm) >  0.80
|    |--- petal width (cm) <= 1.65
|    |    |--- class: 1
|    |--- petal width (cm) >  1.65
|    |    |--- class: 2
```

```
0.9
```

5.4.4 Versiones extendidas: Random Forest

Este método ejecuta diferentes árboles de decisión y se realiza un proceso de votación para elegir cuál es la predicción final. Los RF son métodos de conjunto (ensamble methods) por este motivo. El número de árboles que genera es un parámetro del sistema:

- Dividimos nuestra serie de datos en varios subconjuntos compuestos aleatoriamente de muestras
- Se entrena un modelo en cada subconjunto
- Para cada nodo, elegir aleatoriamente m variables en las cuales basar la decisión. m debe ser < que el número total de variables
- Se combinan todos los resultados de los modelos (por votación)

Estos métodos obtienen mejores resultados en general que los árboles de decisión convencionales, aunque dificultan la comprensión del modelo generado.

Ventajas:

- Generalmente genera resultados muy buenos.
- Fácil de calcular
- Dar estimaciones de qué variables son importantes para clasificar

Desventajas:

- Sobreajusta si hay mucho ruido
- Es más difícil de interpretar que el DT.

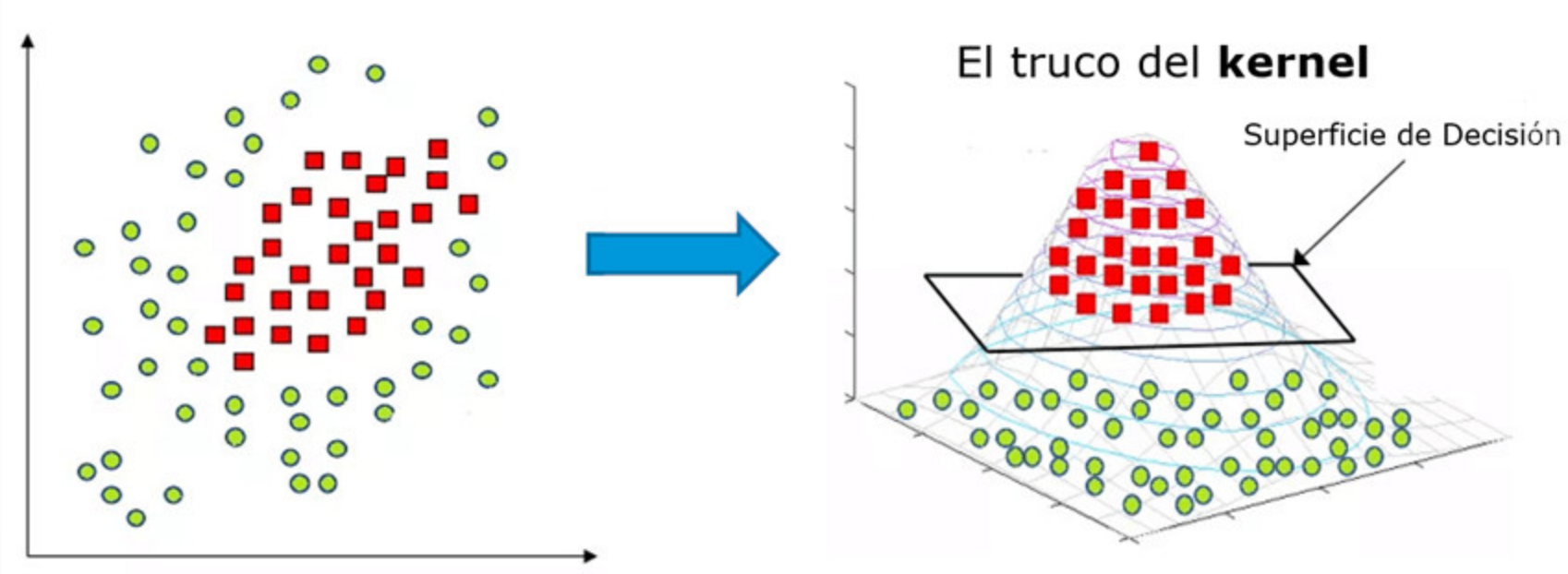
Support Vector Machine

Son un conjunto de algoritmos de aprendizaje supervisado desarrollados por **Vladimir Vapnik** y su equipo en los laboratorios AT&T.

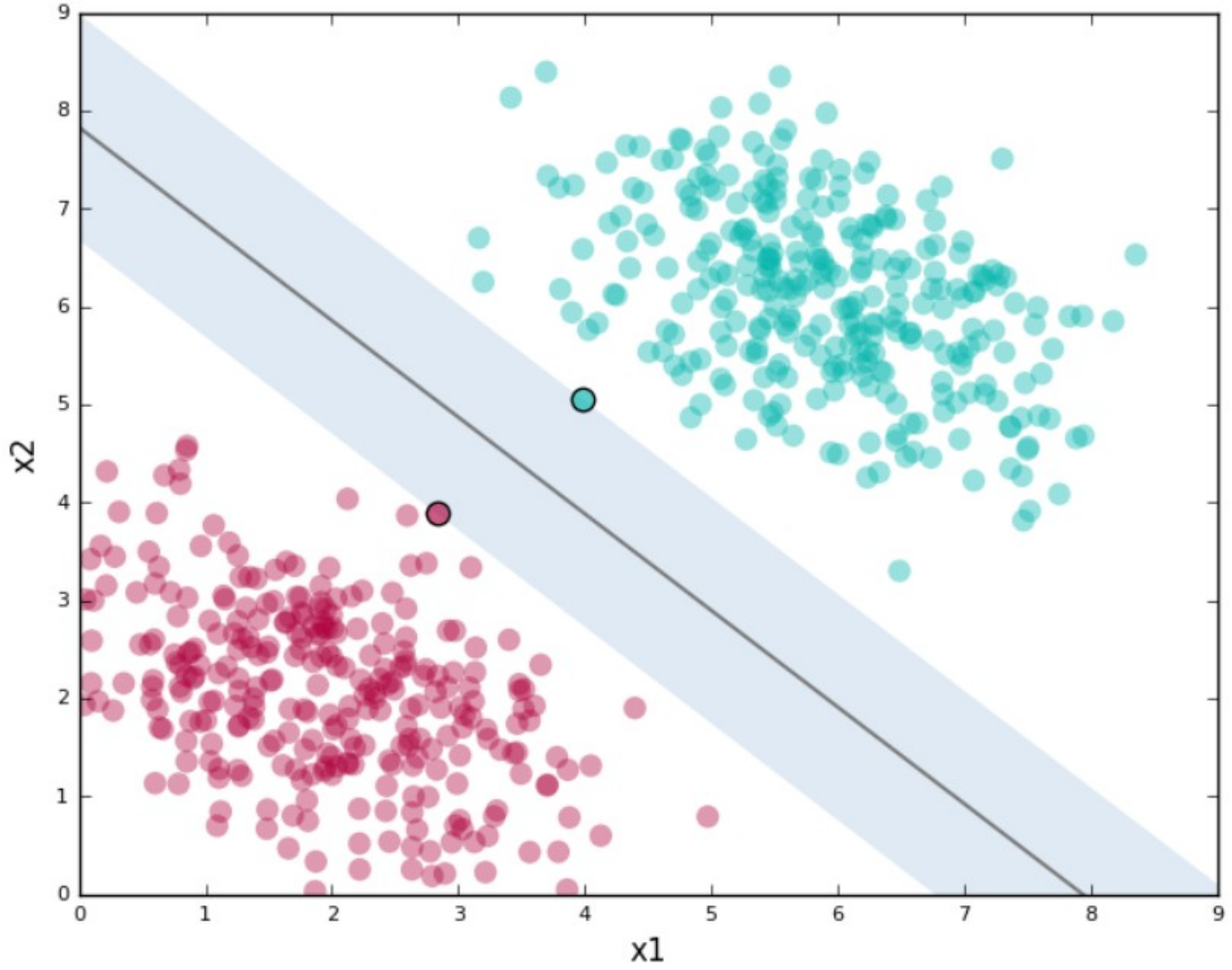
Estos métodos están propiamente relacionados con problemas de clasificación y regresión.

Los vectores de soporte son los puntos que definen el margen máximo de separación del hiperplano que separa las clases. Se llaman vectores porque estos puntos tienen tantos elementos como dimensiones tenga nuestro espacio de entrada.

Hay veces en las que no hay forma de encontrar un hiperplano que permita separar dos clases. En estos casos, decimos que las clases no son linealmente separables. Para resolver este problema podemos usar un **kernel**. El truco del kernel consiste en inventar una dimensión nueva en la que podamos encontrar un hiperplano para separar las clases.



The closest points that identify the lines of the margin are known as *support vectors*



Support vector machine

Logistic regression:

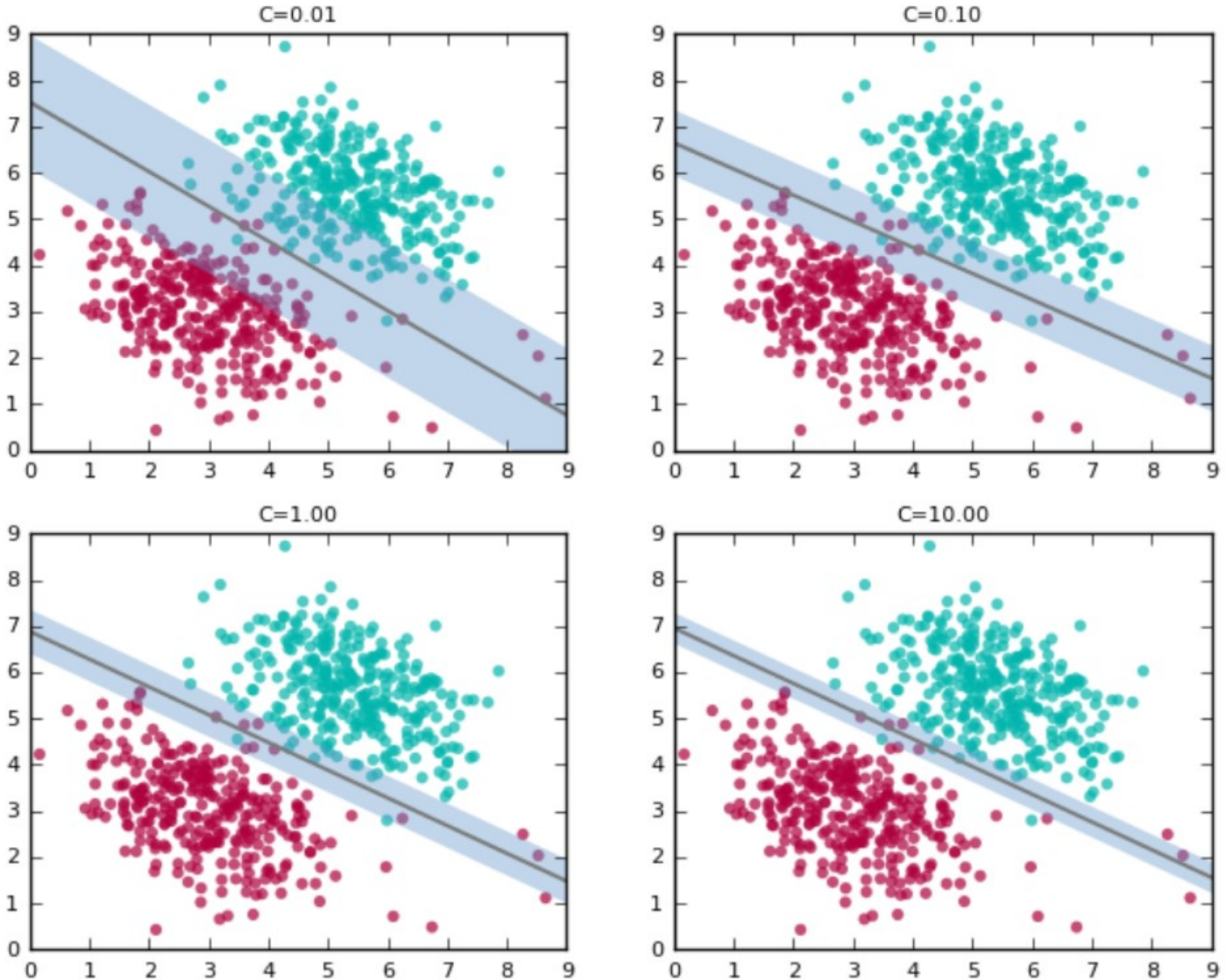
$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\left(-\log h_\theta(x^{(i)})\right) + (1-y^{(i)})\left((-\log(1-h_\theta(x^{(i)}))\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Support vector machine:

$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) cost_0(\theta^T x^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

C controls overfitting



Ventajas

- Eficaz en espacios de grandes dimensiones.
- Todavía eficaz en casos donde el número de dimensiones es mayor que el número de muestras.
- Utiliza un subconjunto de puntos de entrenamiento en la función de decisión (llamada vectores de soporte), por lo que también es eficiente en memoria.
- Versátil: se pueden especificar diferentes funciones del núcleo para la función de decisión. Se proporcionan kernels comunes, pero también es posible especificar kernels personalizados.

Desventajas

- Si el número de características es mucho mayor que el número de muestras evite el exceso de ajuste al elegir las funciones del Kernel y el término de regularización es crucial.
- Los SVMs no proporcionan directamente estimaciones de probabilidad, éstas se calculan utilizando una validación cruzada.

SVM en SKLearn

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
#Fit the model for the data

classifier.fit(X_train, y_train)

#Make the prediction
y_pred = classifier.predict(X_test)
print(y_pred)
model.score(X_test, y_test)
```