Algoritmos de Machine Learning y sus aplicaciones

Trabajo de Fin de Grado

Javier Díaz Bustamante Ussia

5 de agosto de 2015

Índice

Resumen

En este trabajo vamos a estudiar diferentes algoritmos de *Machine Learning* para la clasificación de sucesos. Para comprobar su funcionamiento, los utilizaremos sobre distintas bases de datos, una de reconocimiento de dígitos, una de supervivientes del Titanic y la última de búsqueda del bosón de Higgs.

1. Introducción

En un mundo cada vez más tecnológico, las bases de datos crecen cada día. Cuando alguien entra en una página web, realiza una compra en un comercio, una empresa realiza un estudio de mercado, se celebran unas elecciones, se están almacenando datos. Con este ingente flujo de datos, la física no se podía quedar atrás, hoy en día los grandes aceleradores de partículas manejan al día millones de sucesos que hay que catalogar, clasificar y estudiar, pero la enorme cantidad de estos datos hace imposible su tratamiento *manual*.

Al calor del problema del tratamiento de datos surge el *Machine Learning* (de ahora en adelante *ML*), con algoritmos cada vez más potentes capaces de sacar el máximo partido a estos datos. En este trabajo estudiaremos en concreto cinco de ellos, todos de clasificación (también los hay de regresión). Estos cinco son *Logistic Regresion*, *Naïve Bayes*, *Support Vector Machine*, *Random Forest* y *K-Nearest Neighbors*, explicados en detalle en la seccion 3.

Existen dos tipos de problemas en *ML*, los de clasificación y los de regresión. Estos últimos tratan de predecir el valor de una variable para un conjunto de datos nuevos, como por ejemplo la regresión por mínimos cuadrados. Los problemas de clasificación consisten en tratar de asignar a cada entrada de datos nuevos una clase concreta, pudiendo ser una clasificación binaria (verdadero o falso) o una clasificación multiclase, como por ejemplo un algoritmo de reconocimiento de dígitos. A lo largo de este trabajo, por similitud con el problema del bosón de Higgs, trataremos únicamente con problemas de clasificación binaria.

2. Consideraciones previas

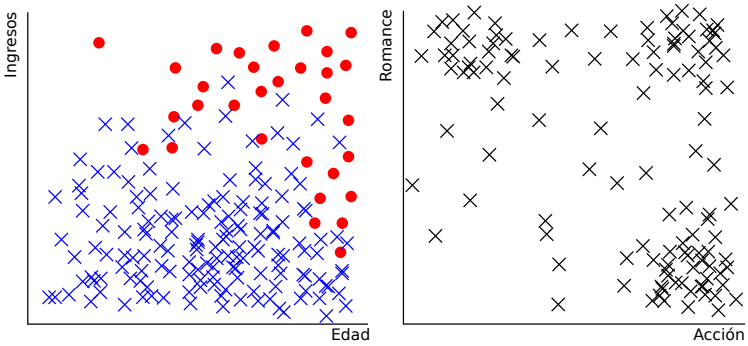
2.1. Aprendizaje supervisado y no supervisado

Antes de explicar cada uno de los algoritmos, vamos a ver algunas características comunes de *ML*. Para empezar, veamos la diferencia entre *aprendizaje supervisado* y *aprendizaje no supervisado*. En *ML*, el término aprendizaje (o entrenamiento) se refiere al análisis de los datos por parte del algoritmo. Éste recibe un conjunto de datos de aprendizaje (el *training set*) y los analiza para tomar una decisión (ya veremos más adelante cómo). La diferencia entre aprendizaje supervisado y aprendizaje no supervisado es que en el primero el conjunto de datos de aprendizaje se encuentra perfectamente etiquetado, mientras que en el no supervisado no lo está.

Por ejemplo, si quisiéramos entrenar un algoritmo para saber a qué tipo de personas les gusta la música clásica, podríamos crear un training set con entradas como: "A Juan Pérez, de 64 años, de clase alta, casado y con carrera universitaria, residente en Madrid, le gusta la música clásica", y otras entradas del estilo de "A Pedro Gutiérrez, de 24 años, clase media, soltero, sin carrera universitaria, residente en Villalpando, no le gusta". Éste sería un problema de clasificación supervisada, y podemos ver un ejemplo de training set ficticio en la gráfica 1a.

Por otra parte, podemos tener datos como por ejemplo el gusto por diferentes tipos de películas de los clientes de un videoclub. A una persona pueden gustarle los romances un 80 %, mientras que las películas de acción sólo un 57 %. A otra, sin embargo, le gustan las primeras al 43 % y las segundas al 96 %. Un ejemplo

de training set podría ser el de la gráfica 1b. En ella, aunque no tengamos clasificadas a las personas, se puede comprobar de forma más o menos nítida que el videoclub tiene tres tipos de clientes distintos.

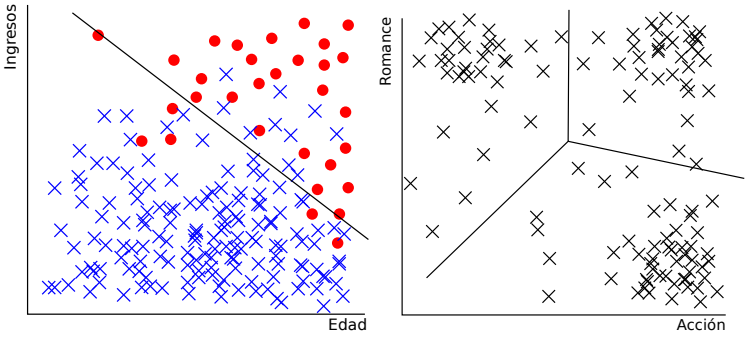


(a) Supervisado. En rojo, los amantes de la música clásica.

(b) No supervisado.

Gráfica 1: Aprendizaje supervisado (a) y no supervisado (b).

Tanto en el aprendizaje supervisado como en el no supervisado, el objetivo de un algoritmo de clasificación es el de elegir una superficie de decisión que separe distintas regiones a las que se asignará cada una de las clases. Estas superficies pueden ser más o menos complejas, dependiendo del algoritmo que se utilice. Un ejemplo de curvas de decisión se recoge en la gráfica 2.



Gráfica 2: Curvas de decisión lineales.

2.2. Métricas

Una vez el algoritmo ha sido entrenado sobre un training set, es hora de probarlo. Para ello se emplea otro conjunto de datos, llamado test set[1], que se clasifica según el algoritmo disponga. En el caso de clasificación binaria, el algoritmo asignará a cada entrada del test set una etiqueta de clase (verdadero o falso, 1 ó 0). Nos interesa saber cómo de preciso ha sido el algoritmo, para lo que disponemos de diversas métricas. Según la etiqueta original de cada entrada

[1]Muchas veces se suele tener sólo un conjunto de datos, que se divide en distintas partes, una el training set, otra el test set, y otra el cross-validation set, que veremos más adelante.

y según la etiqueta predicha por el algoritmo, se pueden dar los casos de la tabla 1.

	0 real	**1 real**
0 predicho	Verdadero negativo (TN)	Falso negativo (FN)
1 predicho	Falso positivo (FP)	Verdadero positivo (TP)

Tabla 1: Posibles casos.

Un buen algoritmo será aquel que prediga 0 para todos los negativos y 1 para los positivos, pero la realidad es que esto casi nunca ocurre. Necesitamos, por lo tanto, un mecanismo para evaluar la bondad de un algoritmo, y eso se consigue con las métricas definidas a continuación[2]:

$$\begin{aligned} \text{Accuracy} &= \frac{TP+TN}{TP+FP+TN+FN} & \text{Recall} &= \frac{TP}{TP+FN} \\ \text{Specificity} &= \frac{TN}{FP+TN} & \text{Precision} &= \frac{TP}{TP+FP} \\ \text{F1-score} &= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \end{aligned} \tag{1}$$

La ecuación *Recall* (sensibilidad) nos da una medida de la exactitud de los casos positivos, mientras que *Specificity* (especificidad) nos da la de los casos negativos. *Precision* (precisión) nos da la exactitud de los casos clasificados como positivos.

Aunque pudiera parecer que *Accuracy* (exactitud) es una buena medida de la bondad de la clasificación, no siempre es así. En los casos de clases sesgadas (*skewed classes*), en las que la proporción de una de las dos clases es mucho mayor que la de la otra, un algoritmo que clasifique cualquier entrada de datos como la clase mayoritaria tendría una gran exactitud, aunque no cumpliría con el objetivo de clasificar correctamente los datos. Por ejemplo, supongamos un problema de detección de fraude en transacciones bancarias en el que nuestro algoritmo deba clasificar como 1 las operaciones fraudulentas. En la realidad hay muchas más transacciones legítimas que fraudulentas, en nuestro ejemplo consideraremos que son el 99 % de las transacciones. Si tenemos un algoritmo que prediga siempre que una transacción es legítima, la exactitud será del 99 %, pero no por ello será un buen algoritmo.

El valor F1 soluciona este problema. Se trata de una media armónica de la precisión y la sensibilidad. A lo largo de este trabajo consideraremos ésta como la métrica para decidir si un algoritmo es mejor que otro.

Para estudiar un algoritmo seguimos unos pasos definidos. Primero, entrenaremos el algoritmo con el training set, tras lo que el algoritmo define una *decisión* que utilizará para clasificar nuevos datos. A continuación, con el algoritmo ya entrenado, procedemos a predecir las clases de cada entrada de datos del test set, con lo que podremos calcular nuestras métricas como vimos en las ecuaciones (1). Al final, tenemos un algoritmo entrenado que generaliza su decisión mejor o peor en función de los valores de las métricas.

El concepto de generalizar viene de que normalmente tenemos un conjunto de datos ya clasificados, que usaremos para entrenar el algoritmo, pero este

[2]Ver [1].

algoritmo lo necesitamos no para clasificar ese conjunto, sino para clasificar nuevos conjuntos de datos que estén aún sin clasificar. Por esto no medimos las métricas sobre el mismo conjunto con el que entrenamos, sino que lo hacemos sobre un conjunto distinto, para ver si el algoritmo no sólo clasifica bien el training set, sino cualquier nuevo conjunto de datos del que dispongamos.

2.3. Cross Validation

La mayoría de los algoritmos de *ML* dependen de unos parámetros que hay que fijar manualmente. Por ejemplo, muchos de ellos utilizan el *parámetro de smoothing*, C, que veremos a continuación. El problema es que los valores de estos parámetros influyen en gran medida en la bondad de la clasificación, y una mala elección de los parámetros nos puede hacer pensar que un algoritmo es malo, cuando la realidad es que lo estamos utilizando mal.

Hasta ahora el procedimiento a seguir era entrenar el algoritmo con el training set, y ver la bondad del mismo con el test set. Ahora, para fijar correctamente estos parámetros, usaremos un tercer conjunto, el Cross Validation set, o CV set. Los pasos a seguir ahora son los siguientes:

Primero, elegimos valores para nuestros parámetros. Con esos valores, entrenamos el algoritmo con el training set. Evaluamos las métricas con el CV set, y volvemos a realizar estos pasos con valores distintos de los parámetros, eligiendo el conjunto de parámetros que mejor prediga las clases del CV set. Por último, con el algoritmo entrenado con los mejores parámetros, evaluamos las métricas en el test set. Estas métricas son las que nos dirán cómo de acertado será nuestro algoritmo al clasificar datos nuevos.

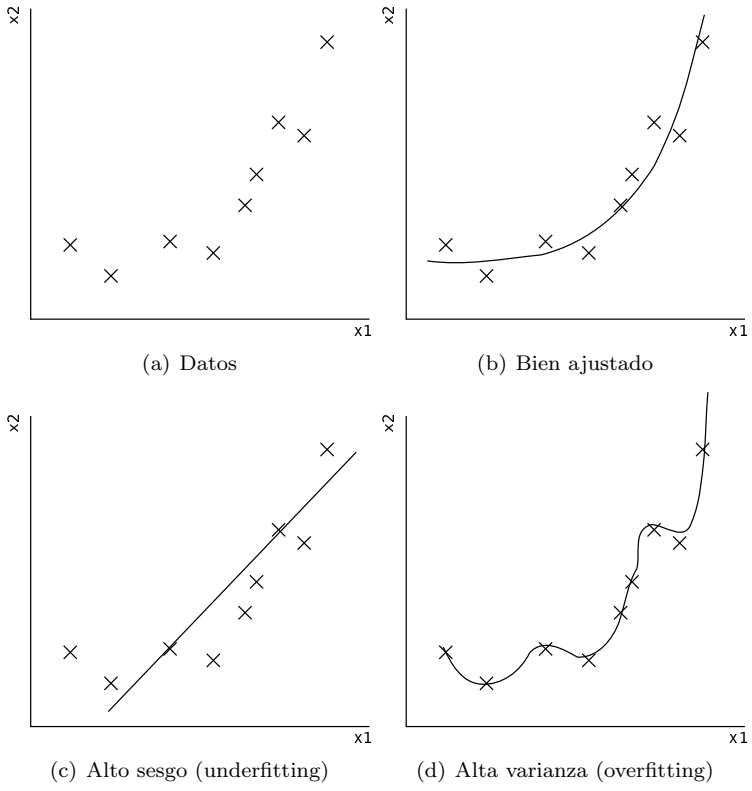
Al igual que antes, el proceso de evaluar métricas se realiza en conjuntos de datos que no hayamos utilizado para entrenar el algoritmo, aunque esta vez tampoco usamos el conjunto que nos ha servido para ajustar los parámetros. Ésto se debe a que, al usar el CV set para ajustar los parámetros, éste ya no nos dice cómo generaliza nuestro algoritmo a datos nuevos, pues el algoritmo final depende de la información que aporta el CV set. Para ver cómo generaliza, tenemos que alimentar nuestro algoritmo con datos que todavía no se hayan usado.

2.4. Varianza, Sesgo y Regularización

Al entrenar un algoritmo nos enfrentamos a dos errores diametralmente opuestos, el *sesgo* y la *varianza*. Tendremos alto sesgo (*underfitting*) cuando nuestra hipótesis es demasiado sencilla o tengamos pocas variables de entrada. La consecuencia es que nuestro algoritmo no generaliza bien los datos nuevos porque tampoco ajusta bien los datos del training set, no somos capaces de extraer información relevante de los datos que tenemos. Para solucionarlo, lo que podemos hacer es tomar datos nuevos con más variables, crear variables derivadas de las que ya tengamos, hacer una hipótesis más compleja o modificar el parámetro de smoothing, que veremos en seguida.

Por el contrario, tendremos alta varianza (*overfitting*) en nuestro algoritmo cuando nuestra hipótesis es demasiado compleja y no tenemos suficientes datos. Nuestro algoritmo no generalizará bien para datos nuevos porque se ciñe demasiado a los datos del training set, perdiendo la tendencia general de estos datos al darle más importancia a las variaciones específicas de este conjunto.

Cometeremos un muy bajo error en los datos del training set, pagando un alto error en el test set. Para solucionarlo podremos tomar más datos, simplificar la hipótesis o modificar el parámetro de smoothing. Podemos ver un claro ejemplo de ambos problemas en la gráfica 3.



(a) Datos (b) Bien ajustado

(c) Alto sesgo (underfitting) (d) Alta varianza (overfitting)

Gráfica 3: Datos para la regresión (a), regresión correcta (b), regresión con alto sesgo (c) y regresión con alta varianza (d).

A continuación explicamos en qué consiste el parámetro de smoothing. Gran parte de los algoritmos se basan en una hipótesis en la que se asigna unos pesos a cada variable (*feature weights*). El algoritmo optimizará estos pesos para que la predicción realizada por la hipótesis coincida lo máximo posible con las etiquetas de cada entrada de datos. Estos pesos, por lo tanto, serán mayores cuanto más influya la variable en la clasificación. El parámetro de smoothing se encarga de controlar la importancia que se da a las variables, con el objetivo de no depender demasiado de ellas, evitando caer en problemas de alta varianza. Una formulación más precisa sería la siguiente:

Sea un algoritmo de clasificación en el que a cada variable se le asocia un peso θ_j, y en el que cada entrada de datos viene dada por el vector $\boldsymbol{x}^{(i)}$, donde $x_j^{(i)}$ sería el valor de la variable j de la entrada i. Cada entrada está clasificada con una etiqueta $y^{(i)}$. El algoritmo tratará de seleccionar el vector de pesos $\boldsymbol{\theta}$ que optimice un error cometido $J(\boldsymbol{\theta})$ (la fórmula concreta de este error dependerá del algoritmo en cuestión). El parámetro de smoothing[3] C se introduce como un

[3] A veces se utiliza el parámetro de regularización $\lambda \propto C^{-1}$.

término adicional al error que se minimiza. Este término suele ser proporcional al inverso de C y a la suma cuadrática de los θ_j, de forma que un valor alto de C respetará más la hipótesis del algoritmo que un valor bajo de C. En resumen, el nuevo error sería:

$$J'(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \frac{1}{C}\boldsymbol{\theta} \cdot \boldsymbol{\theta} \tag{2}$$

Reduciendo C se consigue suavizar la hipótesis del algoritmo, minimizando el vector de pesos $\boldsymbol{\theta}$, reduciendo la varianza del algoritmo. Por otro lado, si aumentamos C permitimos al algoritmo ajustar los pesos más de acuerdo con su hipótesis, permitiendo que ésta sea más compleja y reduciendo el sesgo del mismo. Por lo tanto, el parámetro de smoothing nos ayuda a mejorar nuestro algoritmo, y lo podremos fijar con el CV set, como vimos en el apartado 2.3.

3. Algoritmos

En esta sección estudiaremos diversos algoritmos de clasificación. La mayor parte de los algoritmos siempre producen una hipótesis y una función de error a minimizar. La hipótesis es una aplicación que va del espacio de las variables (*features*) al de las etiquetas (*labels* o *targets*), es decir,

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{Z}/p\mathbb{Z} : \boldsymbol{x} \mapsto y$$

donde n es la dimensión del vector $\boldsymbol{x}$, p es el número de clases y $\boldsymbol{\theta}$ es el vector de pesos. La forma concreta de la hipótesis depende del algoritmo. La función de error, o función objetivo, mide el error cometido por la hipótesis, y será una función que vaya del espacio de los vectores de pesos al de números reales positivos (incluyendo el 0), es decir, $J : \mathbb{R}^q \to \mathbb{R}^+ : \boldsymbol{\theta} \mapsto J(\boldsymbol{\theta})$, donde q es la dimensión del vector de pesos, que la mayor parte de las veces coincide con la dimensión de las variables. La forma concreta de la función objetivo también depende del algoritmo, así que vamos a estudiar los que utilizaremos.

3.1. Logistic Regression

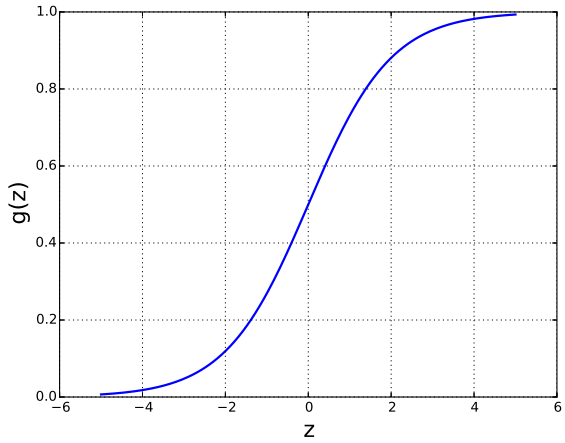
El algoritmo de regresión logística[4] es un algoritmo de modelo lineal, es decir, siempre producirá una superficie de decisión que dependa linealmente de las variables de entrada, aunque si queremos una superficie más compleja sólo tendremos que crear nuevas variables no lineales con las de entrada.

Se llama de regresión logística porque precisamente para la hipótesis hace uso de una función logística, que tiene la forma de la ecuación 3. Ésta es una función sigmoidea, muy comunes en estadística por tener la típica forma de densidad de probabilidad acumulada, como podemos ver en la gráfica 4.

$$g(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

Sea $\boldsymbol{x}$ el vector de variables de entrada, y $\boldsymbol{\theta}$ el vector de pesos, ambos de la misma dimensión. Por convenio, al vector $\boldsymbol{x}$ le añadimos una primera componente, $x_0 = 1$, que tiene su peso correspondiente, θ_0. Esta componente añade

[4] Ver el capítulo 1 de [3]



Gráfica 4: Función logística. La forma de "S" (sigmoidea) es típica de las densidades de probabilidad acumulada en estadística.

un término constante a la combinación lineal de las variables de entrada, como ahora veremos[5]. La hipótesis del algoritmo es (4), donde $g(z)$ es la función logística de (3). Ésta se interpreta como la probabilidad de que, dados el vector $\boldsymbol{x}$ y los pesos $\boldsymbol{\theta}$, la etiqueta correspondiente a esos datos sea "1".

$$\boxed{h_{\boldsymbol{\theta}}(\boldsymbol{x}) = p(y = 1|\boldsymbol{x}; \boldsymbol{\theta}) = g(\boldsymbol{\theta} \cdot \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta} \cdot \boldsymbol{x}}}} \tag{4}$$

Por norma general, predeciremos "$y = 1$" si $h_{\boldsymbol{\theta}}(\boldsymbol{x}) > 0.5$, es decir (ver gráfica 4), si $\boldsymbol{\theta} \cdot \boldsymbol{x} > 0$, por lo que nuestra superficie de decisión, lineal en las variables $\boldsymbol{x}$, será $\{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\theta} \cdot \boldsymbol{x} = 0\}$.

La función de coste nos da el error cometido por la hipótesis, y en el caso de la regresión logística tiene la forma de la ecuación 5.

$$\boxed{J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) \right) + \left(1 - y^{(i)} \right) \log \left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) \right) \right]} \tag{5}$$

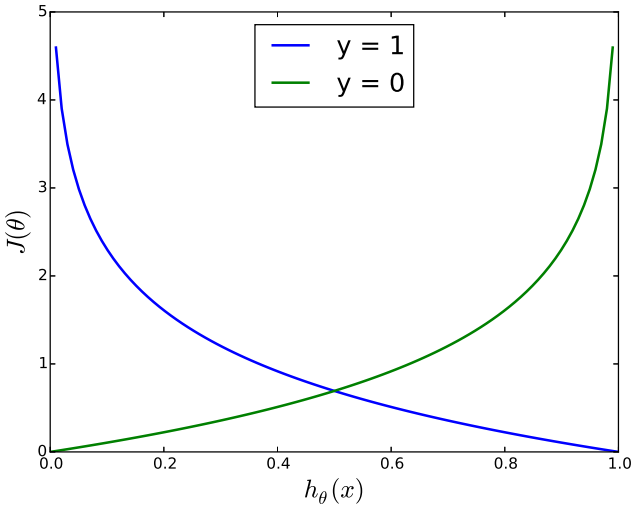
donde m es el número de entradas de datos. Nótese que el primer término del sumatorio corresponde a los datos con $y = 1$, mientras que el segundo es el de los datos con $y = 0$. La función de coste se representa para los casos $y = 1$ y $y = 0$ en la gráfica 5

3.2. Naïve Bayes

El clasificador "Bayes ingenuo" es probablemente uno de los más simples que existen[6]. Se basa en asumir ingenuamente que las variables de las entradas de datos de una clase determinada son independientes, lo que nos puede asustar al

[5] La componente θ_0 no se suele penalizar al hacer la regularización (2), cambiando $\boldsymbol{\theta} \cdot \boldsymbol{\theta}$ por $\sum_{j=1}^{n} \theta_j^2$.

[6] Ver capítulo 20 de [4].



Gráfica 5: Función de error frente a $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ para los casos $y = 1$ y $y = 0$.

oirlo por primera vez, aunque veremos que, a pesar del calibre de la simplificación, funciona sorprendentemente bien aunque las variables no sean realmente independientes[7]. Esta asunción se expresa con la siguiente expresión

$$p(\boldsymbol{x}|C) = \prod_{j=1}^{n} p(x_j|C) \tag{6}$$

donde $\boldsymbol{x}$ es el vector de variables, cuyas componentes son x_j, y C es una determinada clase (en nuestros problemas de clasificación binaria, C será 0 ó 1).

Gracias al teorema de Bayes, sabemos que la probabilidad de que dado el vector de variables $\boldsymbol{x}$, éste sea de la clase C_l es:

$$\boxed{p(C_l|\boldsymbol{x}) = \frac{p(C_l)p(\boldsymbol{x}|C_l)}{p(\boldsymbol{x})}} \tag{7}$$

En general, para un problema de clasificación multiclase, siendo C_l cada una de las clases, el algoritmo asignará a cada entrada nueva de datos la clase C_k que maximice la probabilidad de la ecuación 7. Ésta es la hipótesis del algoritmo. Nótese que, dado un vector $\boldsymbol{x}$, $p(\boldsymbol{x})$ es una constante, por lo que sólo tenemos que preocuparnos por maximizar el numerador, teniendo en cuenta (6).

Para calcular las probabilidades $p(x_j|C)$, hay que tener en cuenta el tipo de distribución que sigue la variable j, dando lugar a diferentes algoritmos, como por ejemplo, el "Naïve Bayes Gaussiano", el "Naïve Bayes Multinomial" o el "Naïve Bayes Binomial", todos ellos nombrados según la distribución de probabilidad de las variables. También se pueden hacer algoritmos de Naïve Bayes mixtos, donde a distintas variables les correspondan distintas distribuciones de probabilidad.

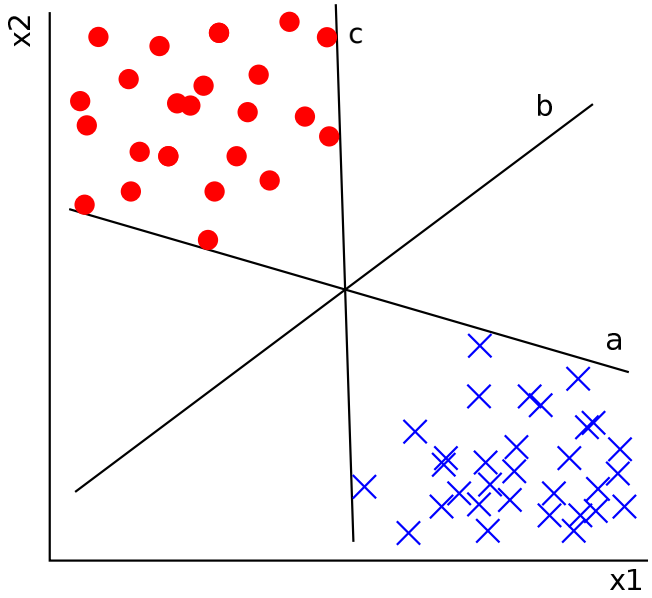
[7] Ver [5].

Las superficies de decisión entre dos clases C_i y C_j serán las que cumplan (8).

$$\{\boldsymbol{x} \in \mathbb{R}^n : p(C_i|\boldsymbol{x}) = p(C_j|\boldsymbol{x}) > p(C_k|\boldsymbol{x}) \quad \forall k \neq i, j\} \tag{8}$$

3.3. Support Vector Machine

Las maquinas de soporte vectorial[8] (*SVM* por sus siglas en inglés) son otro algoritmo de aprendizaje supervisado de *ML*. Muchas veces son llamadas también *large margin separator* o *maximum margin separator*, porque la superficie de decisión que toman es la que más separa los datos del training set. Como vemos en la gráfica 6, todas las líneas separan a la perfección los datos, pero intuitivamente esperamos que la que mejor los separe sea la curva b. ¿Por qué esperamos esto? La respuesta es muy sencilla, esperamos que al tomar nuevos datos, estos sigan la misma distribución que los antiguos, así que las curvas a y c, que tienen puntos muy cercanos, puede que empiecen a clasificar mal alguna entrada de datos. Sin embargo, la curva b, que está más separada de los datos actuales, seguramente no tenga problema al clasificar datos nuevos. Lo que hace que las *SVM* sean tan potentes es que no tratan de minimizar el error empírico, sino de minimizar el error de generalización que podremos cometer al tomar datos nuevos.

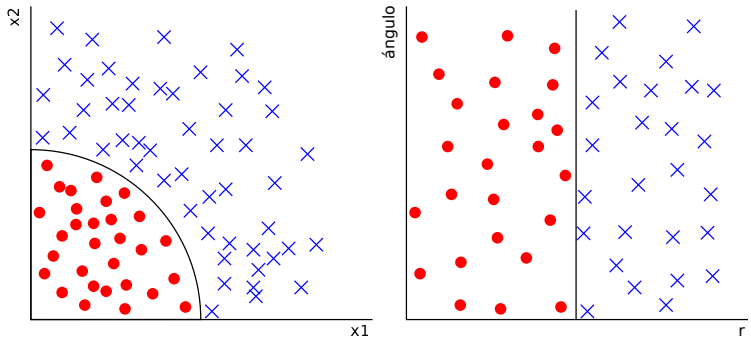


Gráfica 6: Distintas líneas de decisión. La que elegirá una SVM será la b, por ser la que deja mayor margen, minimiza el error de generalización.

Las *SVM* tienen la característica de crear una curva de decisión lineal siempre en las variables que emplea. Esto no debería suponer un problema, pues utilizando *kernels* (que no son otra cosa que cambios de variables) cambiamos

[8] Ver [6], [7], capítulo 10 de [8], capítulo 4 de [9] y capítulo 18 de [4].

nuestro espacio de variables a otro espacio que no sea lineal en el anterior, consiguiendo en el nuevo espacio superficies de decisión no lineales en el antiguo, como vemos en la gráfica 7, en la que se ha cambiado del espacio formado por x_1 y x_2 al formado por la distancia al origen y el ángulo con el eje x_1.



(a) Variables antiguas (b) Variables nuevas

Gráfica 7: Con un cambio de variables (*kernel*) podemos crear superficies de decisión no lineales.

Para la formulación matemática de las *SVM* es importante notar que las etiquetas tradicionalmente no son $y^{(i)} \in \{0, 1\}$, sino $y^{(i)} \in \{-1, +1\}$. Tampoco añadimos el término independiente como una componente más del vector de pesos y la componente $x_0^{(i)}$ del de variables, sino como un término independiente extra, quedando la superficie de decisión como la ecuación 9.

$$\{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\theta} \cdot \boldsymbol{x} + b = 0\} \tag{9}$$

Para conseguir el máximo margen, trataremos de que los puntos más cercanos a (9) cumplan que

$$\begin{aligned} \boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)} + b &= +1 \text{ si } y^{(i)} = +1 \\ \boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)} + b &= -1 \text{ si } y^{(i)} = -1 \end{aligned} \tag{10}$$

por lo que para cualquier punto tendremos que

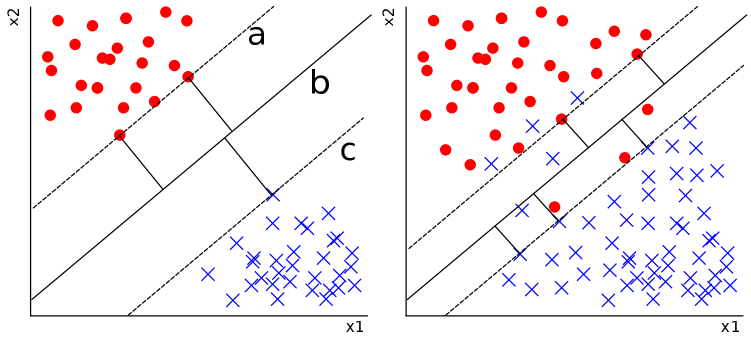
$$y^{(i)} \left(\boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)} + b \right) \geq 1 \quad \forall i = 1, ..., m \tag{11}$$

Los puntos que cumplen que $y^{(i)} \left(\boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)} + b \right) = 1$ se llaman *vectores de soporte*, y son los que dan el nombre al algoritmo. Nuestra hipótesis será

$$\boxed{h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \text{signo}(\boldsymbol{\theta} \cdot \boldsymbol{x} + b)} \tag{12}$$

Si miramos la gráfica 8a vemos el hiperplano de separación y los hiperplanos soporte. Las distancias de cada uno de ellos al origen, nombrados como en la gráfica, son

$$\begin{aligned} d_a &= \frac{|b-1|}{||\boldsymbol{\theta}||} \\ d_b &= \frac{|b|}{||\boldsymbol{\theta}||} \\ d_c &= \frac{|b+1|}{||\boldsymbol{\theta}||} \end{aligned} \tag{13}$$



(a) High margin (b) Soft margin

Gráfica 8: Gráfica **a**: Hiperplano de separación (b), con ecuación $\{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\theta} \cdot \boldsymbol{x} + b = 0\}$ e hiperplanos soporte, (a) con ecuación $\{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\theta} \cdot \boldsymbol{x} + b = +1\}$ y (c) con ecuación $\{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\theta} \cdot \boldsymbol{x} + b = -1\}$. Los vectores de soporte son los más cercanos al hiperplano de separación. Gráfica **b**: Ejemplo de problema de *soft margin*. Hay algunos puntos que se clasifican mal con tal de mantener un margen amplio con la mayoría de los datos, reduciendo el error de generalización.

por lo que la distancia entre los hiperplanos soporte, que es la que hay que maximizar para hacer lo propio con el margen, es

$$d = \frac{2}{||\boldsymbol{\theta}||} \tag{14}$$

Finalmente, tenemos que el algoritmo *SVM* se reduce a maximizar (14) sujeto a (11), que es lo mismo que

$$\boxed{\begin{array}{l} \text{Minimizar} \quad ||\boldsymbol{\theta}||^2 \\ \quad \text{sujeto a } y^{(i)} \left(\boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)} + b \right) \geq 1 \quad \forall i = 1, ..., m \end{array}} \tag{15}$$

Como (15) es un problema de programación cuadrática, y este trabajo no intenta ser un tratado de programación no lineal, la manera de resolver este problema de optimización no se explica aquí, recomendando al lector que quiera profundizar en el tema la lectura de [8].

3.3.1. Margen blando

En gran parte de las bases de datos no podremos separar linealmente las variables (gráfica 8b), por lo que el método de *high margin* fallará. Para estos casos se utiliza el método de *soft margin* (margen blando), introduciendo las variables de holgura h_i tales que todos los puntos deben cumplir que

$$y^{(i)} \left(\boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)} + b \right) \geq 1 - h_i \quad \forall i = 1, ..., m \tag{16}$$

Con esto, el problema de minimización se convierte en (17).

$$\boxed{\begin{array}{l}\text{Minimizar} \qquad ||\boldsymbol{\theta}||^2 + C\sum_{i=1}^{m} h_i \\ \quad \text{sujeto a } y^{(i)}\left(\boldsymbol{\theta}\cdot\boldsymbol{x}^{(i)}+b\right) \geq 1-h_i \;\; \forall i = 1,...,m\end{array}} \tag{17}$$

3.3.2. Kernels

A continuación vamos a explicar los *kernels* que veíamos al principio de esta sección. Éstos son aplicaciones no lineales $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} : (\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) \mapsto K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$. Gracias a los kernels, podemos transformar nuestro espacio de variables $\mathbb{R}^n$ a uno de dimensión superior $\mathbb{R}^{n'}$ en el que se pueda hacer una separación lineal de los datos.

Los kernels tienen la propiedad de que se puede encontrar una transformación $\phi : \mathbb{R}^n \to \mathbb{R}^{n'} : \boldsymbol{x} \mapsto \phi(\boldsymbol{x})$ tal que $K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \phi(\boldsymbol{x}^{(i)}) \cdot \phi(\boldsymbol{x}^{(j)})$. Ésta es la transformación del espacio de las variables que se lleva a cabo. La superficie de decisión será lineal en el nuevo espacio de variables, y con la transformación inversa de ϕ conseguimos una superficie no lineal en las variables antiguas. La gran ventaja de emplear kernels es que no necesitaremos calcular explícitamente $\phi(\boldsymbol{x})$.

Hay muchos tipos de kernels. Los más comunes son: kernel polinomial homogéneo ($K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = (\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)})^d$) e inhomogéneo ($K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = (\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)} + 1)^d$); kernel de función de base radial gaussiano, también llamado *RBF kernel* ($K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \exp\left(-\frac{||\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}||^2}{2\sigma^2}\right)$); kernel lineal, igual que el polinomial homogéneo con $d = 1$, que corresponde a no hacer nada en el espacio de variables. No hay ninguno mejor que otro, cada base de datos tendrá uno de ellos que funcione mejor que los demás, pero en otras bases de datos ese kernel puede ser peor. Parte del "*cocinado*" al preparar un algoritmo *SVM* para una base de datos será encontrar qué kernel y con qué valores de los parámetros (d y σ, por ejemplo) funcionará mejor.

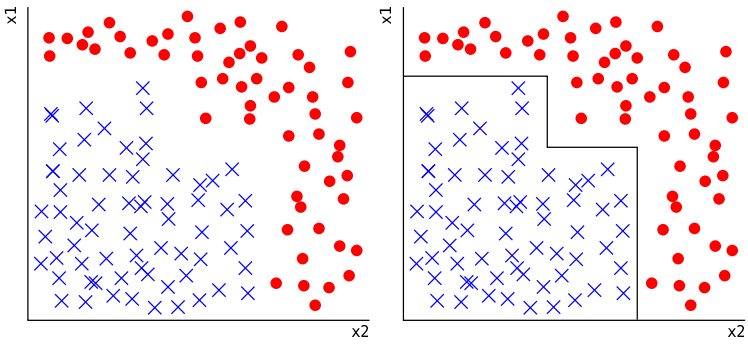
3.4. Random Forest

Los Random Forest son un ejemplo de los llamados *ensemble methods*, en los que se utiliza repetidas veces un algoritmo más sencillo, y se combinan los resultados de estas repeticiones para dar un resultado común. En el caso de los Random Forest lo que se utiliza son árboles de decisión (*decision trees*).

3.4.1. Decision Trees

Podemos ver un ejemplo de base de datos en la gráfica 9a, mientras que en 9b vemos una posible línea de decisión. Ésta puede parecer muy artificial, muy simple, pero funciona bien y es la que hace un árbol de decisión.

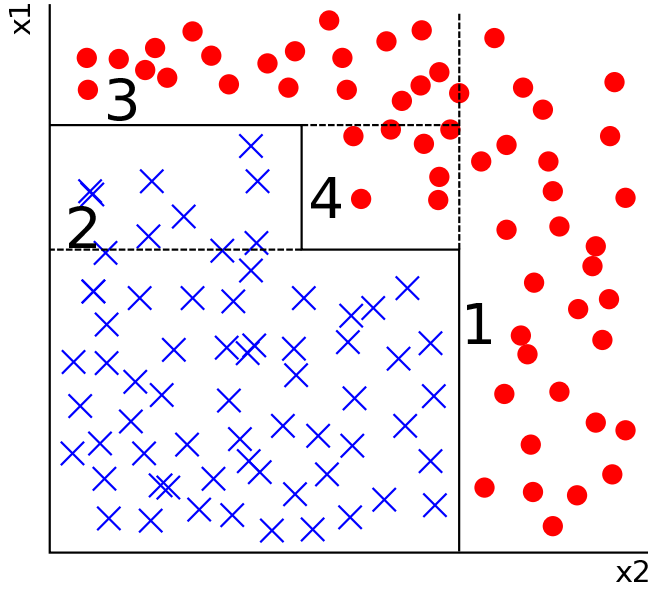
¿Cómo toma esa línea de decisión? El algoritmo consiste en ir separando los datos en regiones. En cada paso separará una región de los datos en dos, y el criterio de separación es sencillamente disminuir la imperfección (desorden) de las regiones. Para poner un ejemplo, en la gráfica 10, primero separaríamos los datos con la línea 1, consiguiendo una región sin imperfección a la derecha de la línea. Luego la línea 2, que parte la región de la izquierda de la línea 1 en dos



(a) Datos (b) Línea de decisión

Gráfica 9: Ejemplo de línea de decisión tomada por un árbol de decisión.

regiones, deja la región de abajo sin desorden. Con dos líneas más conseguimos separar perfectamente los datos.



Gráfica 10: Proceso para encontrar la línea de decisión de un Decision Tree.

Hay varios criterios para separar las regiones. El primero de ellos, la **entropía**, se define por

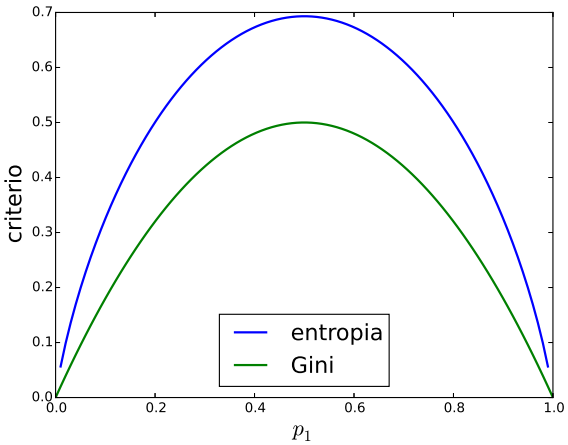
$$-\sum_k p_{mk} \log p_{mk} \tag{18}$$

donde p_{mk} es la proporción, en tanto por uno, de la clase k en la región m. Otro criterio es el criterio **Gini**, definido por

$$\sum_k p_{mk}(1 - p_{mk}) \tag{19}$$

Hay más criterios, pero éstos son los más importantes y usados. El desorden

medido por ambos criterios en función de la proporción de la clase 1 se muestra en la gráfica 11.



Gráfica 11: Desorden en función de p_1.

Vamos a analizar el ejemplo de la gráfica 10. Al principio tenemos una única región con un 0.6 de positivos (puntos rojos) y un 0.4 de negativos (cruces azules). Utilizaremos (19) para calcular las imperfecciones en este ejemplo. En esta primera región, pues, tenemos:

$$0.6 \times (1 - 0.6) + 0.4 \times (1 - 0.4) = 0.48$$

La línea 1 parte la primera región en dos regiones, la de la derecha con 1.0 de proporción de positivos y con el 0.25 del total de la región anterior, y la de la izquierda con proporciones $(0.3, 0.7)$ $((positivo, negativo)$ por comodidad) y el 0.75 de los datos. Tenemos:

$$0.75 \times [0.3 \times (1 - 0.3) + 0.7 \times (1 - 0.7)] = 0.315$$

donde no se han sumado explícitamente las imperfecciones de las regiones bien separadas por ser nulas. La línea 2 parte la última región en una arriba con proporciones $(0.8, 0.2)$ y el 0.4 de la región anterior y en otra abajo con proporciones $(0.0, 1.0)$. Esto nos queda:

$$0.4 \times 0.75 \times [0.8 \times (1 - 0.8) + 0.2 \times (1 - 0.2)] = 0.096$$

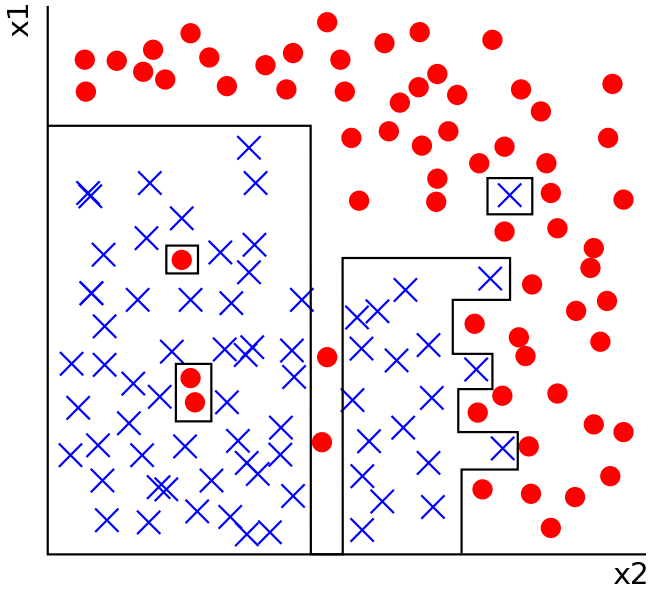
donde hemos tenido que multiplicar la fracción de datos de esta región con la de la anterior. La línea 3, que vuelve a dividir horizontalmente los datos, deja arriba el 0.5 de ellos con proporción $(1.0, 0.0)$ y abajo el otro 0.5 con $(0.4, 0.6)$. Tenemos

$$0.5 \times 0.4 \times 0.75 \times [0.6 \times (1 - 0.6) + 0.4 \times (1 - 0.4)] = 0.072$$

Por último, la línea 4 separa perfectamente los datos de su región, dejándonos con una imperfección nula. Vemos que hemos ido reduciendo la imperfección

(0.48, 0.315, 0.096, 0.072, 0.00) con cada línea. Ésto es lo que hace un árbol de decisión.

Uno de los mayores problemas de un árbol de decisión es su tendencia a sobreajustar los datos. Con unos datos como los usados en el ejemplo anterior no había ningún problema, porque eran fácilmente separables, pero con datos más realistas y complejos, podemos tener una línea de decisión como la de la gráfica 12, con finas franjas para unos pocos datos y regiones aisladas para uno o dos puntos. Las bases de datos reales pueden ser aún más complejas que la de la gráfica, haciendo que la línea de decisión del árbol no tenga realmente ningún sentido general. Una de las maneras de resolver esto es exigir que el algoritmo no divida regiones con pocos datos, o que no deje regiones cerradas con menos de cierto número de datos. Estos pasos requieren de mucho "*cocinado*" para cada base de datos. Otra forma, bastante más extendida, es emplear varios árboles de decisión inicializados de forma aleatoria, creando un **Random Forest**.



Gráfica 12: Los árboles de decisión son propensos al sobreajuste.

3.4.2. Random Forest

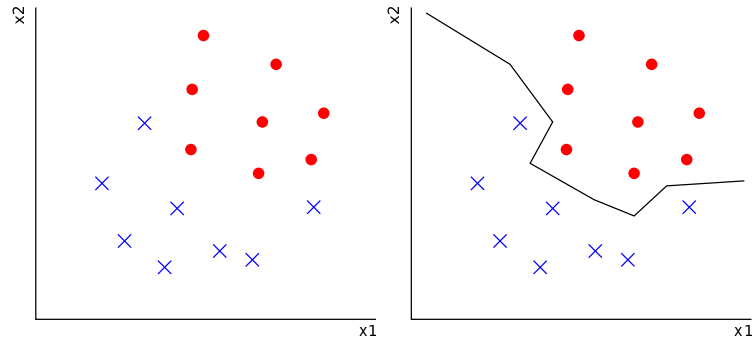
Blablabla

3.5. K-Nearest Neighbors

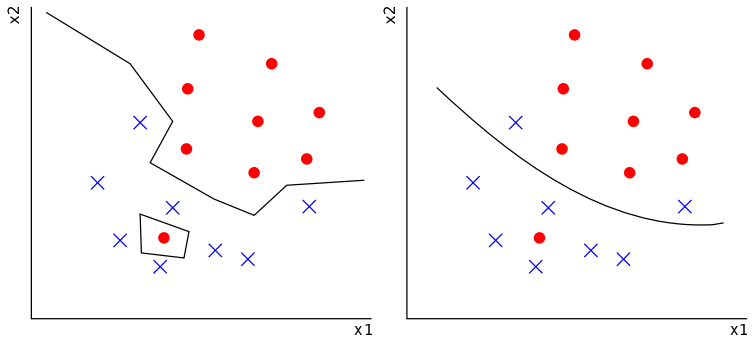
Llegamos a otro algoritmo más sencillo aún que el Naïve Bayes, si esto puede ser posible. El K-Nearest Neighbors (KNN) se explica casi únicamente al leer el nombre[9]. Se trata de un algoritmo que asigna a cada entrada nueva de datos la clase más común entre las K entradas clasificadas más cercanas (del training set), utilizando una métrica dada (normalmente la euclídea). Como parámetros

[9] Ver [11] y [12]

a ajustar de este algoritmo, tenemos únicamente la métrica a utilizar y el número de vecinos.



(a) Datos (b) 1 vecino



(c) 1 vecino, overfitting (d) 3 vecinos, ya no hay overfitting

Gráfica 13: Con pocos vecinos, el algoritmo es propenso al overfitting.

En la grafica 13, vemos la sencillez del algoritmo. Hay que tener cuidado con el número de vecinos, porque si escogemos pocos podemos caer en overfitting. También es importante antes de entrenar el algoritmo, hacer un *feature scaling*[10], de manera que no haya una variable cuyas distancias entre datos sean mucho menores que el resto de variables, dando a todas ellas la misma importancia.

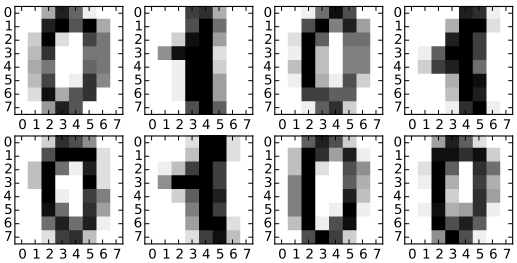
4. Aplicaciones

En esta sección veremos cómo se aplican los anteriores algoritmos a bases de datos reales. Para cada una de ellas explicaremos las variables de que se disponen, las peculiaridades de los datos y los resultados que extraemos.

[10]El *feature scaling* cambia todas las variables para que sus valores se encuentren en el intervalo [0,1]. Es importante hacerlo tanto para éste algoritmo como para el Support Vector Machine (sobre todo el kernel gaussiano), aunque nosotros lo hacemos para todos los algoritmos por comodidad.

4.1. Reconocimiento de dígitos

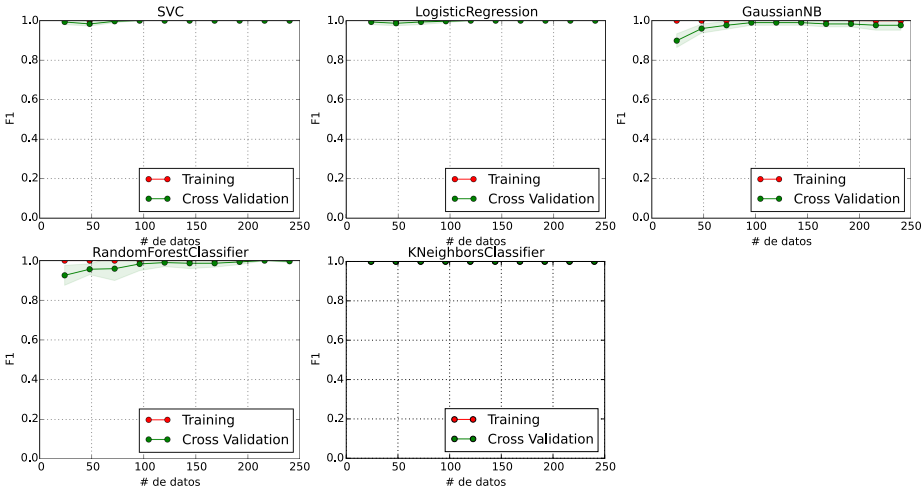
Gracias al paquete `sklearn` de Python disponemos de una serie de bases de datos para hacer pruebas. La que vamos a usar es la de reconocimiento de digitos, que consiste en una serie de imágenes de 8×8 píxeles, cada una representando un dígito del 0 al 9. Por similitud con la base de datos del Higgs, nosotros solo utilizaremos las imagenes con digitos 0 y 1, transformando el problema en uno de clasificación binaria.



Gráfica 14: Ejemplos de imágenes de la base de datos.

Las variables de esta base de datos son las intensidades de cada uno de los píxeles, con valores comprendidos entre 0 y 16, leídos por filas desde arriba. Tenemos por tanto 64 variables para cada una de las 360 imágenes, que son nuestras entradas de datos.

Procedemos a estudiar los datos. Lo primero que se suele hacer es una curva de aprendizaje, en la que representamos la bondad de cada algoritmo en función del número de datos estudiados. Esto se hace para ver si necesitamos tomar más datos, o si por el contrario podemos despreciar algunos, ganando en velocidad de cálculo. La curva de aprendizaje de esta base de datos se recoge en la gráfica 15.



Gráfica 15: Curvas de aprendizaje para la base de datos de los dígitos.

En la gráfica 15 podemos ver cómo los algoritmos de Logistic Regression, Support Vector Machines y K-Nearest Neighbors consiguen un buen valor F1 aún con pocos datos, y que la curva del training set y la del CV set son prácticamente paralelas, así que usar más datos no va a ser mejor, y de hecho podemos usar menos datos. El resto de algoritmos presentan un comportamiento más común, en el que la curva del training set va reduciendo su valor F1 al aumentar el número de datos, mientras que los del CV set van aumentando al aumentar los datos. El mejor número de datos a utilizar sería aquel para el que las dos curvas sean prácticamente paralelas, así que según las gráficas podríamos utilizar sólo 100 entradas de datos. En la realidad, al ser ésta una base de datos bastante pequeña, los algoritmos no tardan mucho en ser entrenados, así que usaremos la base de datos entera.

Realizamos ahora el Cross Validation. La malla inicial de parámetros, común a todas las bases de datos, es la siguiente: Para el SVC (Support Vector Classifier) usamos el kernel RBF con los valores de C 1, 10, 100, 1000, y los valores de γ 0.0001, 0.001, 0.01, 0.1, 1.0, y el kernel lineal con los mismos valores de C. Como el Naïve Bayes no depende de parámetros no podemos hacer sobre él ningún cross validation. Para el Logistic Regression usamos los mismos valores de C. Para el Random Forest usamos 5, 10 y 15 árboles, para el mínimo de datos requerido para poder separar una rama usamos 2, 4, 6, 8 y 10 datos, y para el mínimo de datos por hoja usamos 1, 2, 3, 4 y 5 datos. Para el KNN (K-Nearest Neighbors) usamos como número de vecinos 2, 3, 4, 5 y 6.

Los mejores parámetros que hemos encontrado son los siguientes: Para el SVC, el kernel RBF con $C = 1$ y $\gamma = 0.001$; para el Logistic Regression, $C = 1$, para el Random Forest, 15 árboles, con 6 datos como mínimo para separar una rama y con 1 dato como mínimo para cada hoja; y para el KNN 2 vecinos. Las métricas medidas por cada algoritmo se recogen en la tabla 2.

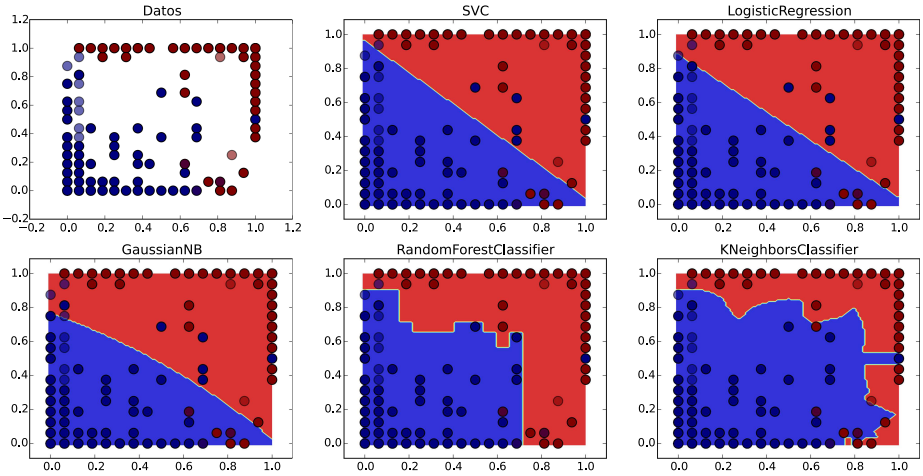
Algoritmo	Accuracy	Precision	Recall	F1-Score
Gaussian NB	0.9667	0.9688	0.9688	0.9688
SVC	0.9500	0.9677	0.9375	0.9524
Logistic Regression	1.0000	1.0000	1.0000	1.0000
Random Forest	1.0000	1.0000	1.0000	1.0000
KNN	1.0000	1.0000	1.0000	1.0000

Tabla 2: Métricas de los dígitos para los distintos algoritmos.

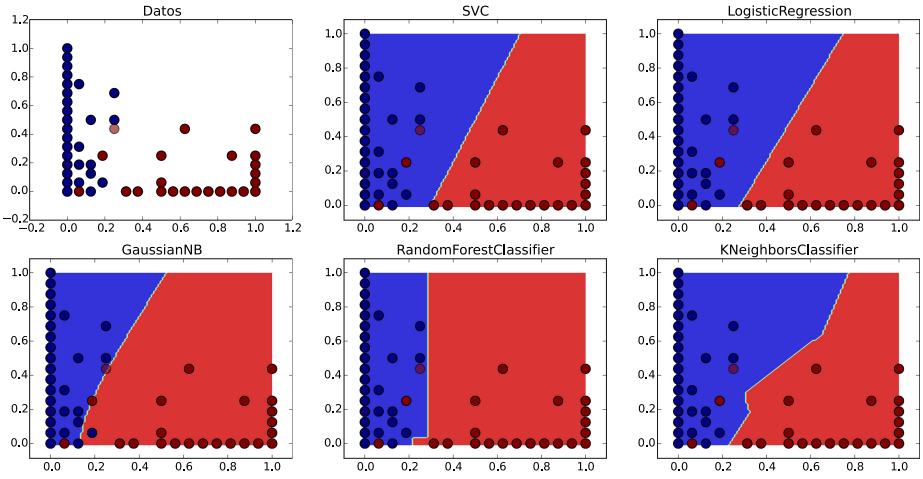
Hemos llegado a un problema (bendito problema): que tres de nuestros algoritmos funcionan perfectamente. Para elegir cuál es el que nos compensa usar, habrá que ver si buscamos rapidez, robustez (que aguante bien posibles alteraciones en los datos), optimizar la memoria... El mejor algoritmo dependerá de la aplicación real que hagamos de él, pero este ejemplo nos sirve para mostrar que no hay un algoritmo mejor que otro, que, dependiendo de la base de datos, pueden funcionar todos igual de bien (el SVC, que ha sido el peor, tiene un valor F1 de 0.95, bastante alto, y seguramente haciendo un ajuste más fino de los parámetros conseguiríamos un valor mejor).

Para ilustrar lo que hemos conseguido, podemos realizar las gráficas con las variables más influyentes que se han encontrado, consiguiendo las gráficas 16 y 17, en las que se ha hecho un feature scaling para que las variables estén en el

intervalo [0,1].



Gráfica 16: Curvas de decisión para los diferentes algoritmos utilizando los valores de los píxeles 21 (x) y 28 (y)



Gráfica 17: Curvas de decisión para los diferentes algoritmos utilizando los valores de los píxeles 29 (x) y 47 (y)

4.2. Supervivientes del Titanic

Esta base de datos se encuentra en la página web del Kaggle **METER UN LINK AQUI**, una web en la que se proponen concursos abiertos de Machine Learning. Este ejemplo concreto se trata de un problema que proponen como iniciación para los nuevos usuarios, y consiste en conseguir detectar qué pasajeros sobrevivieron al accidente. Las variables con las que contamos son las siguientes: la categoría del billete, que es una variable categórica (1ª, 2ª o 3ª clase); el sexo, variable binaria; la edad; el número de parientes y esposas a

bordo (sin contar padres ni hijos); el número de padres e hijos a bordo; el precio del billete; y el lugar de embarque, variable categórica con valores C, S y Q **Rellenar los nombres de los lugares de embarque...**.

4.3. Búsqueda del bosón de Higgs

5. Conclusiones

Referencias

[1] S. Bhattacharyya, S. Jha, K. Tharakunnel y J. C. Westland. *Decision Support Systems*, **50**, 602-613 (2011).

[2] S. Jha, M. Guillen, J. C. Westland. *Expert System with Applications*, **39**, 12650-12657 (2012).

[3] D. W. Hosmer, S. Lemeshow, R. X. Sturdivant. Applied Logistic Regression, *John Wiley & Sons* (2013).

[4] S. Russel, P. Norvig. Artificial Intelligence: A Modern Approach, *Prentice Hall* (2010).

[5] I. Rish. An empirical study of the naive Bayes classifier, *IBM Research Report* **No sé cómo citar esto...** (2001).

[6] J. E. Hernández, S. Salazar. *Scientia et Technica*, **31**, 47-52 (2006)

[7] C. Cortes, V. Vapnik. *Machine Learning*, **20**, 273-297 (1995)

[8] V. Vapnik. Statistical Learning Theory, *John Wiley & Sons* (1998).

[9] J. L. Martínez Pérez. *Comunicación con computador mediante señales cerebrales. Aplicación a la tecnología de la rehabilitación* (2009). Tesis doctoral de la ETSII (UPM). **Tampoco sé si se cita así...**

[10] J. R. Quinlan. *Machine Learning*, **1**, 81-106 (1986).

[11] T. M. Cover, P. Hart. *IEEE Transactions on Information Theory*, **13**, 21-27 (1967)

[12] T. M. Cover. *IEEE Transactions on Information Theory*, **14**, 21-27 (1968)