Practica2.pdf

Angie_Floro

Aprendizaje Automatico

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

APRUEBA TODO

Trabajos universitarios por encargo

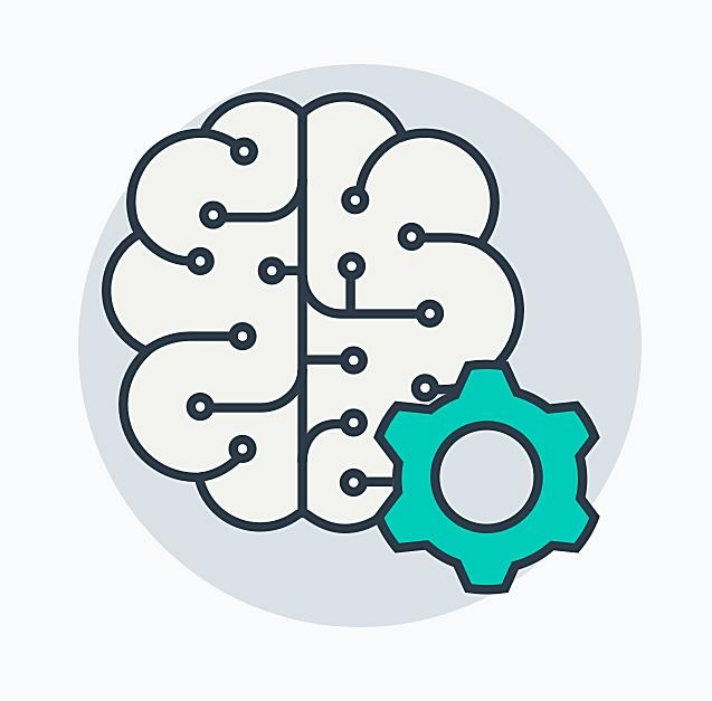
¿Atascado con tu TFG?

Aquí tenemos la solución

+ DE 5.000
PROFESORES EXPERTOS

PRESUPUESTO
SIN COMPROMISO
EN 24 HORAS

apruebatodo.com



Trabajo 2: Programación

APRENDIZAJE AUTOMÁTICO

Ángeles Caballero Floro | 22 de Abril de 2019

Contenido

Ejercicio sobre la complejidad de H y el ruido

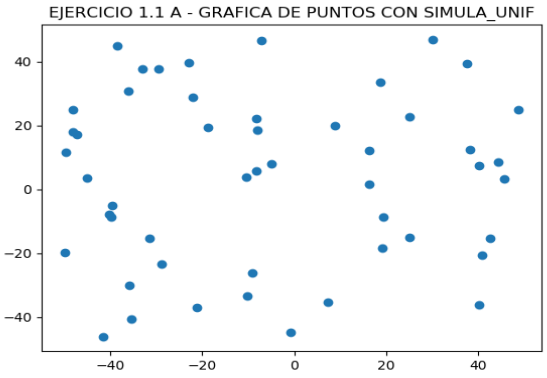
Este ejercicio se centra en la dificultad que supone la aparición de ruido a la hora de elegir la función más adecuada.
El ruido supone que tenemos, en nuestra población, un porcentaje de elementos que han sido mal clasificados, es decir, que esos valores han sido mal recogidos, transcritos, procesados... En nuestro caso, realizaremos una simulación controlada de ruido intercambiando las etiquetas de un porcentaje preciso de la población que ha sido seleccionado de forma aleatoria.

1-Graficas con simula_unif() y simula_gaus()

Este apartado no admite mucha explicación. Únicamente se nos pedía hacer uso de las funciones simula_unif() y simula_gaus() proporcionadas por los profesores para crear:

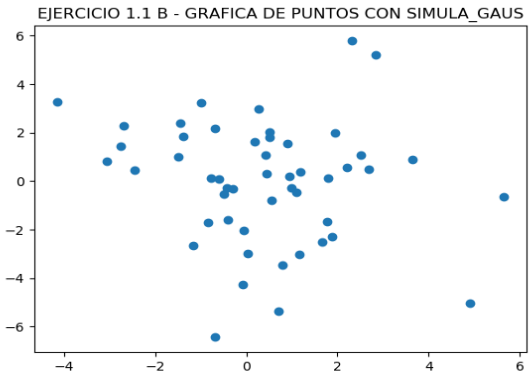
SIMULA_UNIF()

Una nube de 50 puntos en dos dimensiones (contenida en un cuadrado de [-50, 50] x [-50, 50]) con distribución uniforme.



SIMULA_GAUS()

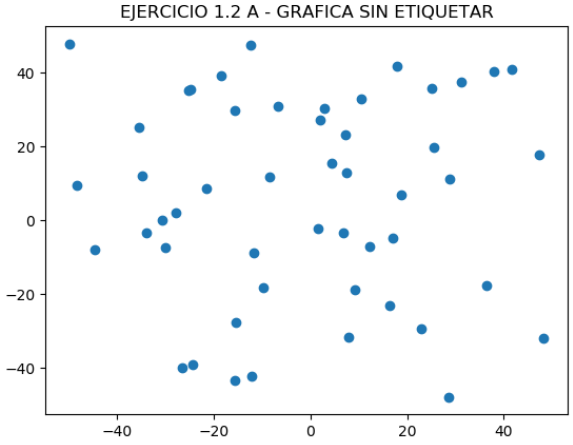
En este caso, se nos pide exactamente lo mismo que en el apartado anterior (una nube de 50 puntos en dos dimensiones contenida en un cuadrado de [-50, 50] x [-50, 50]) pero en este caso, con distribución gaussiana.



2-Introducción al ruido:

ETIQUETADO SIN RUIDO

Para este apartado, se nos pedía generar una nueva muestra uniformemente distribuida, con las mismas proporciones que la población del apartado anterior. A esta, la he llamado X (conjunto de datos) en la implementación de código de python.



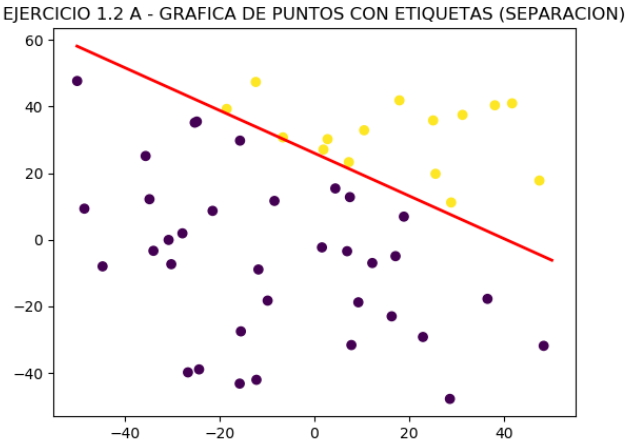
El siguiente paso por llevar a cabo consistía en etiquetar esos datos. Para ello utilizamos como referencia la función f(x,y,a,b), donde entendemos como x e y las coordenadas de esos datos (x = coordenada sobre el eje X e y = coordenada sobre el eje Y).

$$f(x, y, a, b) = y - ax - b$$

Por otro lado, a y b son la pendiente y el termino independiente (respectivamente) de la recta que divide el cuadrado que contiene la nube de datos. Para trazar esa linea y, por tanto, conocer los valores de las constantes a y b, he hecho uso de la función **simula_recta(),** también proporcionada por los profesores.

En conclusión, el criterio de clasificación de la población es:

- $si \;\; f(x, y, a, b) < 0 \;\; entonces \;\; etiqueta = -1$
- $si \;\; f(x, y, a, b) \geq 0 \;\; entonces \;\; etiqueta = 1$



Como podemos observar en la gráfica, todos los datos están correctamente clasificados con respecto a la linea que ha sido trazada para usarse como referencia.

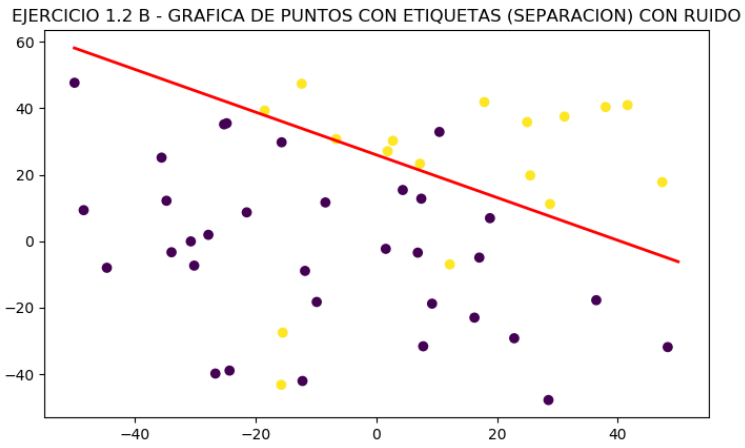
ETIQUETADO CON RUIDO

Una vez tenemos nuestros conjuntos X (conjunto de datos) e Y (conjunto de etiquetas), se nos pide que simulemos un 10% de ruido uniforme en la población creada., es decir un un 10% en la población positiva (amarillo en la gráfica) y otro 10% en la población negativa(morado en la gráfica). Para ello, he creado una función auxiliar llamada **ruido()** que divide la población en dos subpoblaciones separadas (positiva y negativa) y los índices de cada una de ellas que van a ser invertidos para introducir el ruido especificado.

POBLACIÓN	TAMAÑO	[10%]
POSITIVOS (+)	15	1
NEGATIVOS (-)	35	3

Según los datos generados aleatoriamente, tras introducir el ruido si generamos la gráfica, deberíamos tener un punto en la sección positiva (amarilla) que tiene la etiqueta cambiada y por tanto es morado, y tres puntos de la zona negativa (morada) que han cambiado su etiqueta y ahora son amarillos.

Estos puntos pueden cambiar, ya que se genera de forma aleatoria, pero mientras que conservemos los mismos conjuntos X e Y, la proporción se va a mantener. Un ejemplo sería el siguiente:



3-Ruido y clasificación con otras funciones

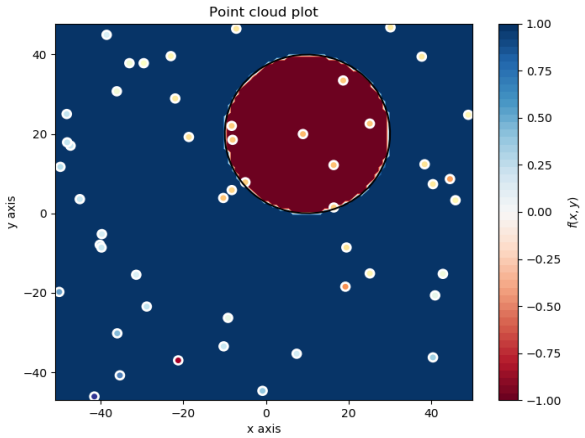
Este último apartado consiste en repetir el procedimiento del anterior, es decir:

1- Clasificamos los elementos de X según f(x,y)
2- Una vez clasificados correctamente, introducimos ruido en la población
3- Pintamos la gráfica resultante

FUNCION 1

$$f(x,y) = (x-10)^2 + (y-20)^2 - 400$$

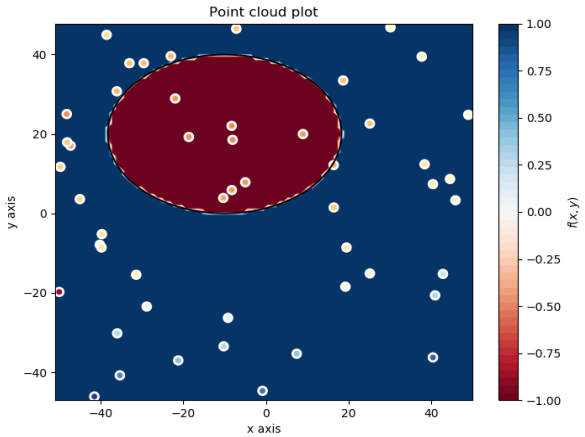
POBLACIÓN	TAMAÑO	[10%]
POSITIVOS (+)	9	0
NEGATIVOS (-)	41	4



FUNCION 2

$$f(x,y) = 0.5(x+10)^2 + (y-20)^2 - 400$$

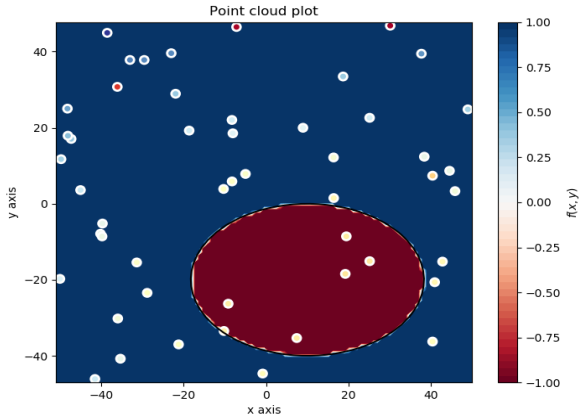
POBLACIÓN	TAMAÑO	[10%]
POSITIVOS (+)	36	3
NEGATIVOS (-)	14	1



FUNCION 3

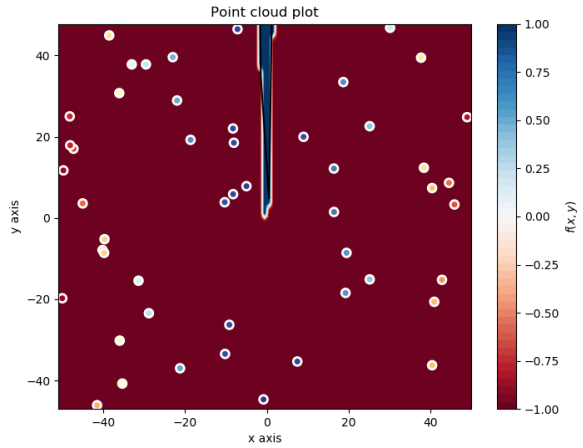
$$f(x,y) = 0.5(x-10)^2 - (y+20)^2 - 400$$

POBLACIÓN	TAMAÑO	[10%]
POSITIVOS (+)	38	3
NEGATIVOS (-)	12	1



FUNCION 4

$$f(x,y) = y - 20x^2 - 5x + 3$$



Este caso (Función 4), es algo distinto a los anteriores ya que la función de clasificación ha etiquetado todos los elementos como que pertenecen a la misma clase, con lo cual, no podemos introducirle ruido a la muestra.

La resolución de este ejercicio es que no importa si una función es más compleja o no, a la hora de la clasificación de datos. Lo que importa es que se adapten a las restricciones del problema. Las funciones más complejas son útiles cuando no se puede separar los datos linealmente, y no lo son tanto en el caso contrario.

Modelos lineales

1 - Algoritmo Perceptrón

En este algoritmo, pretendemos minimizar el error actualizando w (w = w +yx).
Es bastante parecido al SGD implementado en la práctica anterior, pero en este caso η = 1 y el gradiente, vendría dado por yx.

Assigment Rule

$w=w+y_i x_i$ si	$signo(w^T+x_i)$ ¡=$signo(y_i)$
nada	en cualquier otro casS

El algoritmo en sí continuaría iterando mientras que alguno de los datos cumpla la primera condición o (para evitar bucles infinitos) hasta que llegue al máximo número de iteraciones especificado.

Pseudo-code

Given thedata set $\mathbf{x}_n$, y_n,n=1,2,⋯,N
Step.1: Fix w_{ini} = 0
Step.2: Iterate on the D-samples improving the solution:

repeat
  For each $\boldsymbol{x}_i \in \mathcal{D}$ do
    if: $sign\ \boldsymbol{w}^T\boldsymbol{x}_i \neq y_i$ then
      update w: $w_{new}= w_{old}+ y_i x_i$
    else
      continue
  End for
Until No changes in a full pass on $\mathcal{D}$

Una vez implementado el código, se nos pedía ejecutar esta función para el conjunto de elementos clasificados con y sin ruido.

Sin ruido

1 – Si el punto de entrada es w = [0,0,0]

Peso inicial	**Nº de iteraciones**
[0 0 0]	15

2 – Si tenemos diez puntos de partida generados aleatoriamente (con valores entre 0 y 1)

Peso inicial			**Nº de iteraciones**
[**0.90853515**	0.62336012	0.01582124]	8
[**0.92943723**	0.69089692	0.99732285]	8
[**0.17234051**	0.13713575	0.93259546]	10
[**0.69681816**	0.06600017	0.75546305]	11
[**0.75387619**	0.92302454	0.71152476]	10
[**0.12427096**	0.01988013	0.02621099]	17
[**0.02830649**	0.24621107	0.86002795]	11
[**0.53883106**	0.55282198	0.84203089]	8
[**0.12417332**	0.27918368	0.58575927]	3
[**0.96959575**	0.56103022	0.01864729]	8

El número medio de iteraciones generado por los diez casos anteriores es 9.4
Elegir un buen punto de partida (como el que se encuentra sombreado en amarillo en la tabla anterior), supone que nos ahorremos tiempo de computo esperando a que la función converja. En este caso, prácticamente cualquiera de los puntos aleatorios es mejor que el punto de partida inicializado a 0, con lo que podríamos considerar el usar siempre un punto de parida aleatorio para comenzar la ejecución del algoritmo.

Con ruido

1 – Si el punto de entrada es w = [0,0,0]

Peso inicial	Nº de iteraciones
[0 0 0]	5000

2 – Si tenemos diez puntos de partida generados aleatoriamente (con valores entre 0 y 1)

Peso inicial			Nº de iteraciones
[**0.90853515**	0.62336012	0.01582124]	5000
[**0.92943723**	0.69089692	0.99732285]	5000
[**0.17234051**	0.13713575	0.93259546]	5000
[**0.69681816**	0.06600017	0.75546305]	5000
[**0.75387619**	0.92302454	0.71152476]	5000
[**0.12427096**	0.01988013	0.02621099]	5000
[**0.02830649**	0.24621107	0.86002795]	5000
[**0.53883106**	0.55282198	0.84203089]	5000
[**0.12417332**	0.27918368	0.58575927]	5000
[**0.96959575**	0.56103022	0.01864729]	5000

En este caso, el número medio de iteraciones generado por los diez casos anteriores es 5000.

Para hacer más obvia la comparación, decidí almacenar los valores aleatorios generados para el apartado anterior, en lugar de generar unos nuevos. De esta forma puedo comparar mejor las tablas, aunque la conclusión sería la misma: como los datos no están bien clasificados (ruido), el perceptrón no para por si solo, sino que recurre a la condición de parada que se refiere al número máximo de iteraciones.

Se puede deducir entonces, que la función no llega a converger nunca si tiene ruido.

2 - Regresión logística

La primera parte del ejercicio nos pedía implementar el código del algoritmo de Regresión Logística con Gradiente Descendente Estocástico. Para ello haremos uso de funciones auxiliares como **sigmode():**

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

Pseudo-code sgdRL

Given thedata set $\mathbf{x}_n$, y_n,n=1,2,⋯,N,dif_min,tasa

Step.1: Fix $w_{ini} = 0$

Step.2: Iterate on the D-samples improving the solution:

repeat

permutate x y

For each $\boldsymbol{x}_i \in \mathcal{D}$ do

update w: $w_{new} = w_{old}$ - tasa* $\frac{-y_i x_i}{1+e^{-y_i w^T x_i}}$

End for

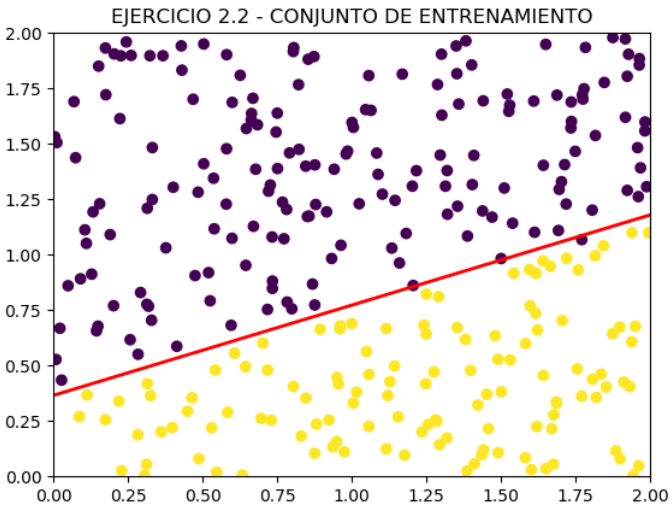
update delta = $|| w_{old} - w_{new}) ||$

Until delta< dif_min

Uso del algoritmo

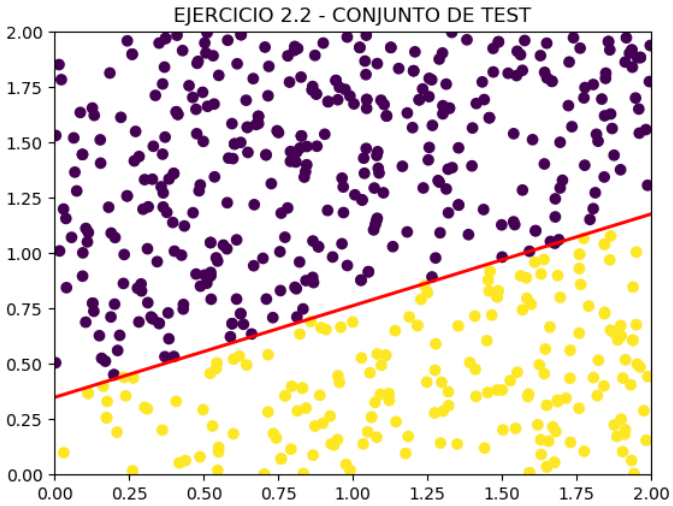
Para empezar, nos piden encontrar la función que nos permita separa de forma adecuada los datos, por eso, el primer paso es dejar que el algoritmo "aprenda" es decir:

1- Dividir los conjuntos de datos y etiquetas en los subconjuntos:
   a. x_train, y_train: Que usaremos para calcular w
   b. x_test y_test: Que usaremos para comprobar si la funcion es correcta
2- Calcular w con el conjunto de datos y etiquetas de entrenamiento (train)



Donde la linea roja es función g que pretendíamos calcular

3- Comprobamos si la función es correcta o no para clasificar los datos. Calculamos el error sobre los conjuntos test y w



En este caso (y debido a las especificaciones del enunciado), mi conjunto de entrenamiento tiene 500 elementos y mi conjunto de prueba tiene los 1000 restantes de una población de 1500. Para llevarlo a cabo, he creado una funcion auxiliar (ver ejercicio de Python)

Calcular el error logístico

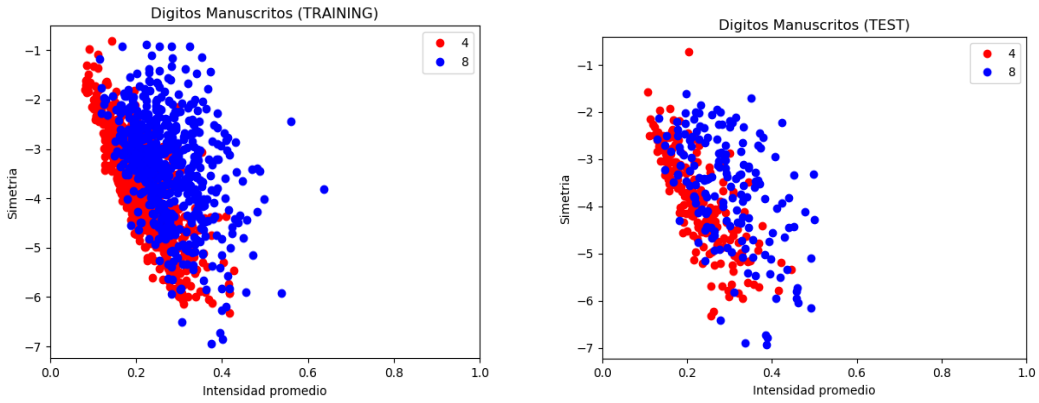
$$Err(w) = \frac{1}{N}\sum_{i=0}^{N} \ln(1 + e^{-y_i w^T x_i})$$

Ein: [0.04896076] <- Calculado con los conjuntos de entrenamiento
Eout: [0.04979472] <- Calculado con los conjuntos de prueba

Un valor tan bajo de los errores supone que la función lineal que hemos calculado clasifica los datos casi de forma perfecta.

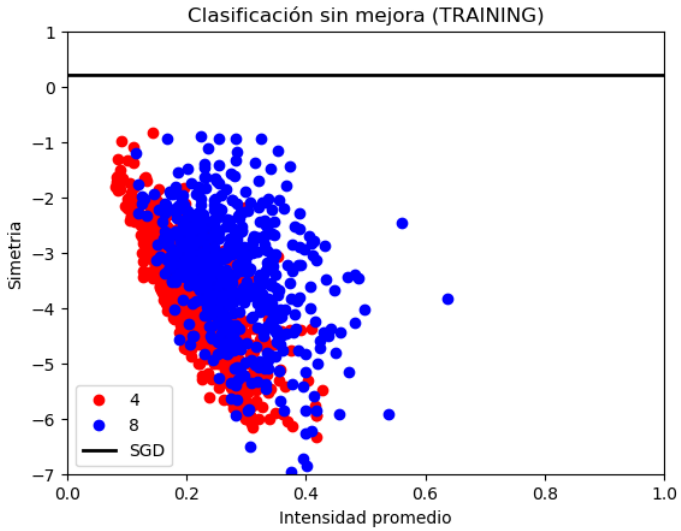
Bonus

Para este ultimo ejercicio, el objetivo era entender el funcionamiento del algoritmo pocket, que es una variante del algoritmo perceptrón que hemos implementado en el ejercicio anterior.
Al igual que con la regresión logística, debemos calcular una función g (mediante los conjuntos de entrenamiento) que consiga clasificar de forma efectiva las muestras que se añadan al conjunto con posterioridad.



Observando los gráficos, es fácil llegar a la conclusión de que los conjuntos van a tener un error muy alto, ya que prácticamente se superponen.

Este algoritmo comienza a funcionar con una solución del problema previamente calculada, en lugar de empezar desde 0. Yo he decidido calcular dicha solución con el algoritmo sgd implementado para la practica anterior.

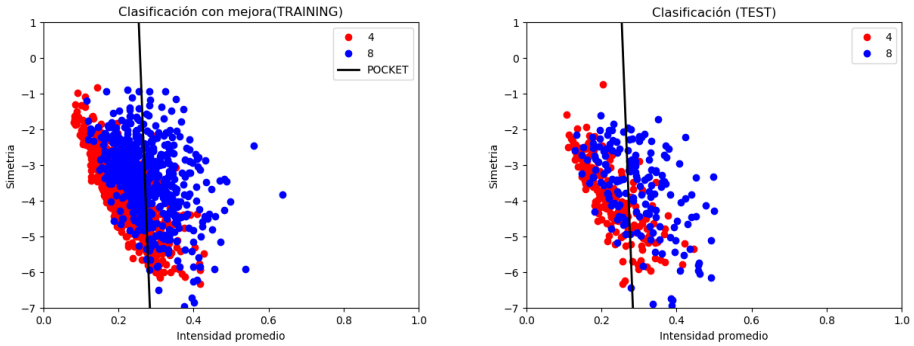


Solo he dejado que el algoritmo iterase 1 vez, con lo cual, es obvio que se trata de una muy mala solución solo mirando la gráfica anterior, pero para ser mas concienciados, he sacado los errores (E_{in}, E_{out}) que se producirían con esa función:

- E_{in}: [[0.98713973]]
- E_{test}: [[0.99041475]]

La clasificación va a fallar casi con toda probabilidad.

A partir de ese punto, podemos utilizar el algoritmo pocket con el que obtendremos con toda seguridad, una función que clasifique mejor los datos.



De nuevo, resulta obvio que, pese a que los datos se clasifican mejor, el error va a ser alto. Las muestras se superponen y no es posible separarlos de forma efectiva mediante funciones lineales. Pese a todo, los errores de la clasificación han disminuido al mejorar el valor de g (w):

- E_{in}: [[0.8258418]]
- E_{test}: [[0.82176545]]