

# Aprendizaje Automático

-

## 2020/2021

### Practica 2

1. **Complejidad de H y el ruido**
  - 1.1. **Gráficas de las nubes de puntos simuladas**
    - 1.1.1. *Distribución uniforme*
    - 1.1.2. *Distribución gaussiana*
  - 1.2. **Influencia del ruido en la complejidad de la clase de funciones**
    - 1.2.1. *Dataset etiquetado y su función*
    - 1.2.2. *Añadiendo ruido a las etiquetas*
    - 1.2.3. *Definiendo la frontera con distintas funciones*
2. **Modelos lineales**
  - 2.1. **Algoritmo del perceptrón**
    - 2.1.1. *Aplicación de PLA-Pocket sobre dataset sin ruido*
    - 2.1.2. *Aplicación de PLA-Pocket sobre dataset con ruido*
  - 2.2. **Regresión Logística**
    - 2.2.1. *Aplicación de SGD-RL sobre dataset*
    - 2.2.2. *Experimento*
3. **Bonus**
  - 3.1. **Clasificando dígitos con Pseudo-Inversa y mejorando con PLA-pocket**
    - 3.1.1. *Pseudo-Inversa*
    - 3.1.2. *PLA-pocket*
  - 3.2. **Cotas de  $E_{out}$** 
    - 3.2.1. *Cota basada en  $E_{in}$*
    - 3.2.2. *Cota basada en  $E_{test}$*

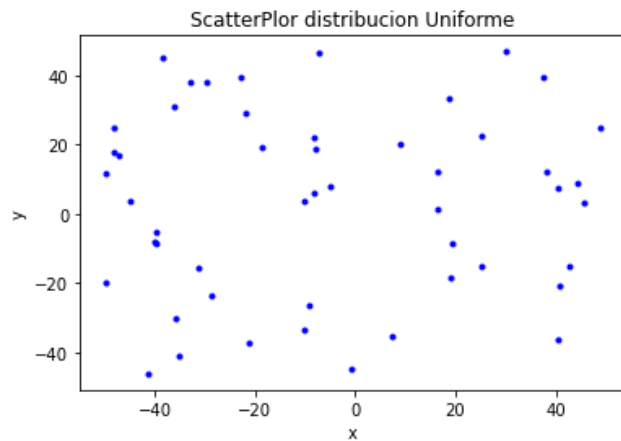
## 1.-Complejidad de H y el ruido

### 1.1.-Gráficas de las nubes de puntos simuladas

Vamos a dibujar distintas nubes de puntos en dos dimensiones con distintas distribuciones de probabilidad

#### 1.1.1.-Distribución uniforme

Utilizando la función `simula_unif(N, dim, rango)` generamos 50 puntos ( $N=50$ ) en dos dimensiones ( $dim=2$ ) en el rango  $[-50,50]$  ( $rango=[-50,50]$ ) con una distribución uniforme. Se puede observar como no hay ninguna tendencia ni distribución respecto a un punto como después ocurrirá en la distribución gaussiana.



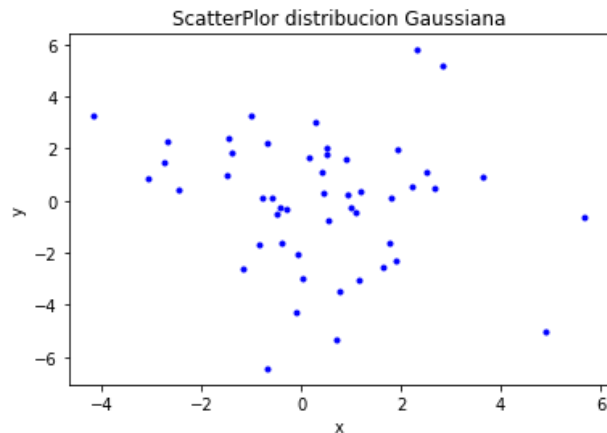
#### 1.1.2.-Distribución gaussiana

Utilizando la función `simula_gaus(N, dim, sigma)` generamos 50 puntos ( $N=50$ ) en dos dimensiones ( $dim=2$ ) con una distribución gaussiana

Sobre eje  $x$ :  $\sigma = 5$   
Sobre eje  $y$ :  $\sigma = 7$

( $\sigma=[5,7]$ )

Se puede observar como los puntos siguen estando dispersos, pero ahora hay más puntos en torno al cero de ambos ejes y cuanto más nos acercamos a los extremos menos cantidad de puntos hay. Los puntos tienen más probabilidad de aparecer cercanos al 0 de los ejes que de estar en un extremo del gráfico, muy alejados del 0

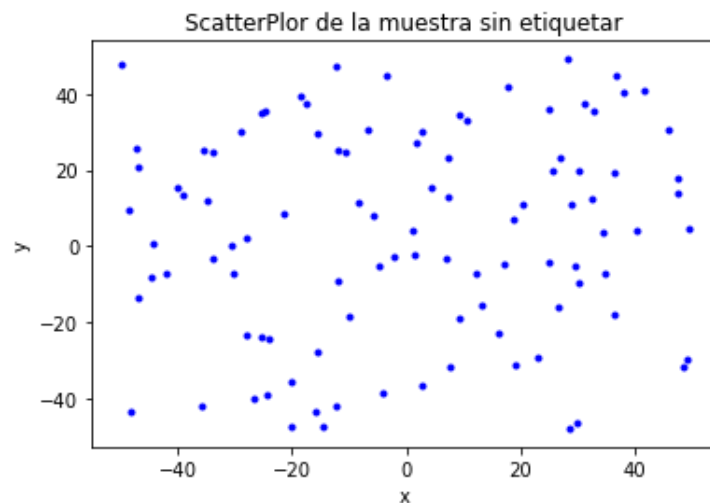


## 1.2.-Influencia del ruido en la complejidad de la clase de funciones

Vamos a ver cómo afecta al error de distintas funciones la introducción de ruido.

### 1.2.1.-Dataset etiquetado y su función

Utilizando la la función `simula_unif(N, dim, rango)` generamos 100 instancias, en el rango  $[-50,50]$  con una distribución uniforme sobre dos dimensiones

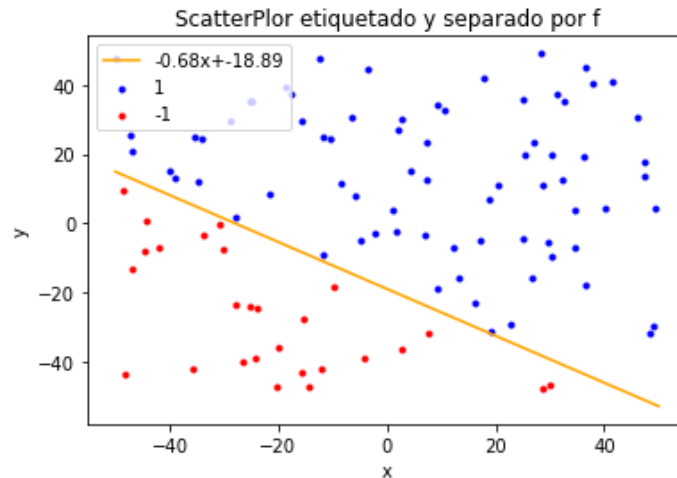


Creamos de manera aleatoria una recta, con la función `simula_recta(rango)`, función que calcula aleatoriamente los valores a y b para obtener una función de la forma:

$$f(x) = ax + b$$

Con la función anterior vamos a etiquetar los puntos con valores 1 o -1 en función de su posición respecto a  $f(x)$ .

$$\begin{cases} 1 & \text{if } y - ax - b \geq 0 \\ -1 & \text{if } y - ax - b < 0 \end{cases}$$



### 1.2.2.-Añadiendo ruido a las etiquetas

Vamos a añadir un 10% de ruido en cada una de las clases, es decir de los puntos azules, un 10% pasarán a ser rojos, y de los puntos rojos, un 10% pasarán a ser azules.

Para hacer esto simplemente cogemos los índices de las etiquetas que valen 1, los barajamos, y calculamos cuántas de esas etiquetas van a pasar a -1

$$\text{Etiquetas a cambiar} = (\text{porcentaje} * \text{Numero de unos}) / 100$$

Y lo único que falta es iterar sobre “Etiquetas a cambiar” numero de elementos del vector de índices de unos barajados, y cambiar su valor a menos uno.

Haciendo lo mismo que antes con las etiquetas que valen menos uno ya tendremos el ruido generado.

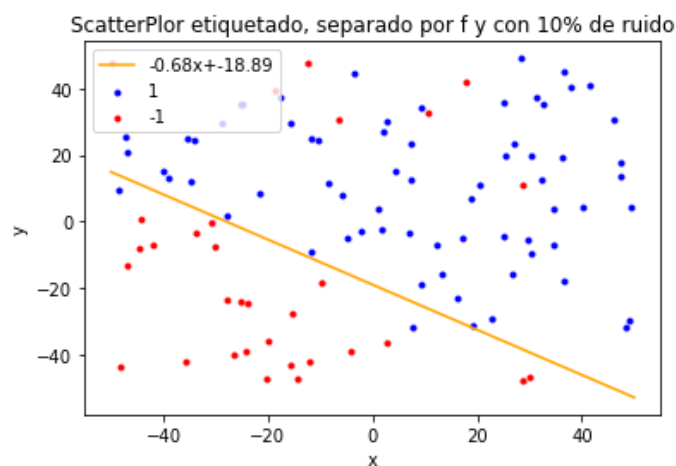
```
Aniadimos ruido a la muestra
Numero de unos: 73
Numero de cambios: 7
Numero de menos unos: 27
Numero de cambios: 2

Confusion matriz y=ax+b
[[66 7]
 [ 2 25]]
```

Número de azules: 73 por lo que se modifican 7 azules a rojos

Número de rojos: 27 por lo que se modificarán 2 rojos a azules

La matriz de confusión la usaremos posteriormente, y explicare como la he creado.



La recta ha clasificado 7 puntos como “1”, cuando realmente tendrían que ser “-1” y 2 puntos como “-1” cuando realmente tendrían que ser “1”. Por lo que clasifica mal 9 puntos.

### 1.2.3.-Definiendo la frontera con distintas funciones

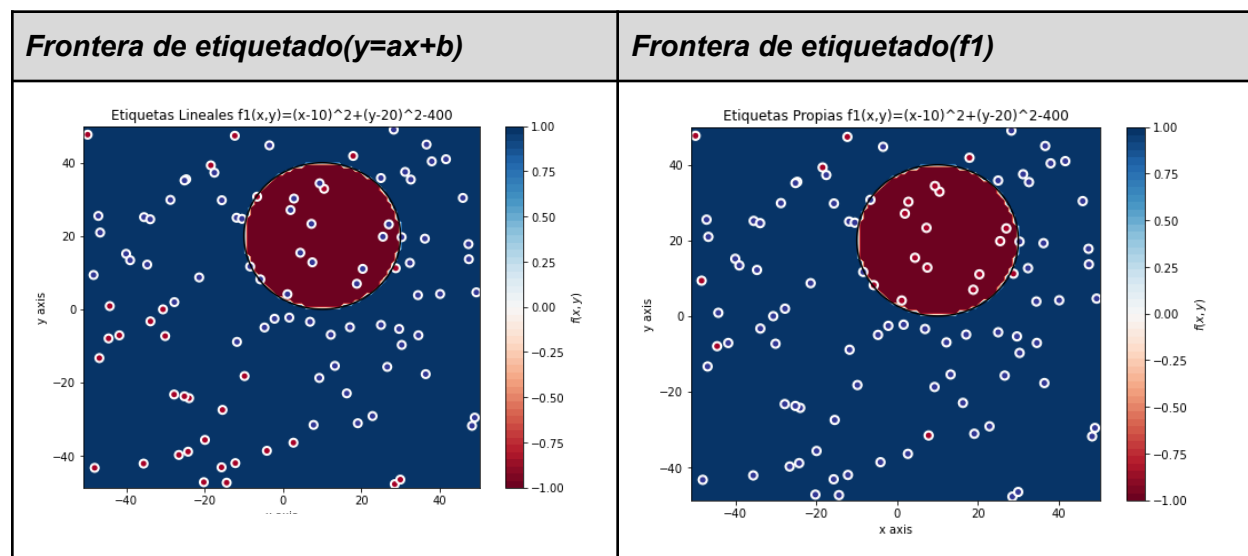
Siendo  $g(x)$  la clase que predice nuestro modelo para la instancia  $x$  y  $f(x)$  la etiqueta real de  $x$ . Una matriz de confusión es lo siguiente:

	$f(x)=1$	$f(x)=-1$
$g(x)=1$	Número de unos correctamente clasificados	Número de instancias clasificadas como unos, pero realmente son menos unos
$g(x)=-1$	Número de instancias clasificadas como menos uno pero realmente son unos	Número de menos unos correctamente clasificados

Para construirla simplemente hay que iterar sobre una muestra, ver lo que clasifica nuestro modelo( $g(x)$ ) y ver su etiqueta real( $f(x)$ ) y en función de ambos sumo uno en una casilla u otra.

Vamos a aumentar la complejidad de las funciones y observar cómo clasifican la muestra generada en el apartado anterior(Columna de la izquierda), así como la misma muestra etiquetada por las propias funciones y con un 10% de ruido(Columna de la derecha)

$$f1(x, y) = (x - 10)^2 + (y - 20)^2 - 400$$



	$f(x)=1$	$f(x)=-1$
$g(x)=1$	56	30
$g(x)=-1$	12	2
Total mal clasificados	42	

	$f(x)=1$	$f(x)=-1$
$g(x)=1$	78	8
$g(x)=-1$	1	13
Total mal clasificados	9	

Podemos observar cómo para la muestra etiquetada por la función lineal clasifica 30 puntos como “1” cuando realmente son “-1”, así como ha clasificado 12 puntos como “-1” cuando realmente son “1”. Y para la muestra etiquetada por ella misma, obtiene un error de un 10% en cada clase, un total de 9 errores

$$f_2(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$$

Frontera de etiquetado( $y=ax+b$ )

Etiquetas Lineales  $f_2(x,y)=0.5(x+10)^2+(y-20)^2-400$

Frontera de etiquetado( $f_2$ )

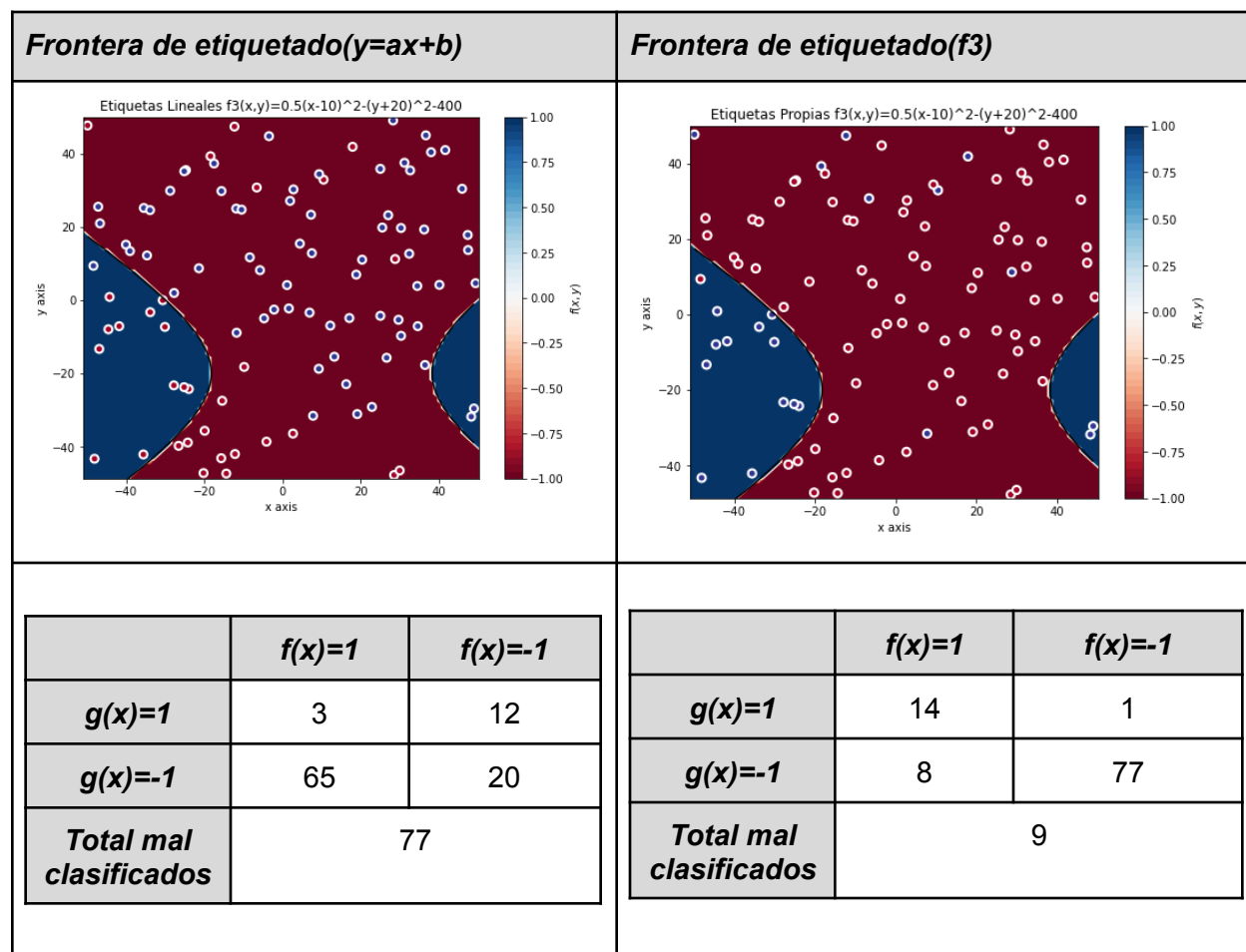
Etiquetas Propias  $f_2(x,y)=0.5(x+10)^2+(y-20)^2-400$

	$f(x)=1$	$f(x)=-1$
$g(x)=1$	48	30
$g(x)=-1$	20	2
Total mal clasificados	50	

	$f(x)=1$	$f(x)=-1$
$g(x)=1$	71	7
$g(x)=-1$	2	20
Total mal clasificados	9	

Podemos observar cómo para la muestra etiquetada por la función lineal clasifica 30 puntos como “1” cuando realmente son “-1”, así como ha clasificado 20 puntos como “-1” cuando realmente son “1”. Y para la muestra etiquetada por ella misma, obtiene un error de un 10% en cada clase, un total de 9 errores

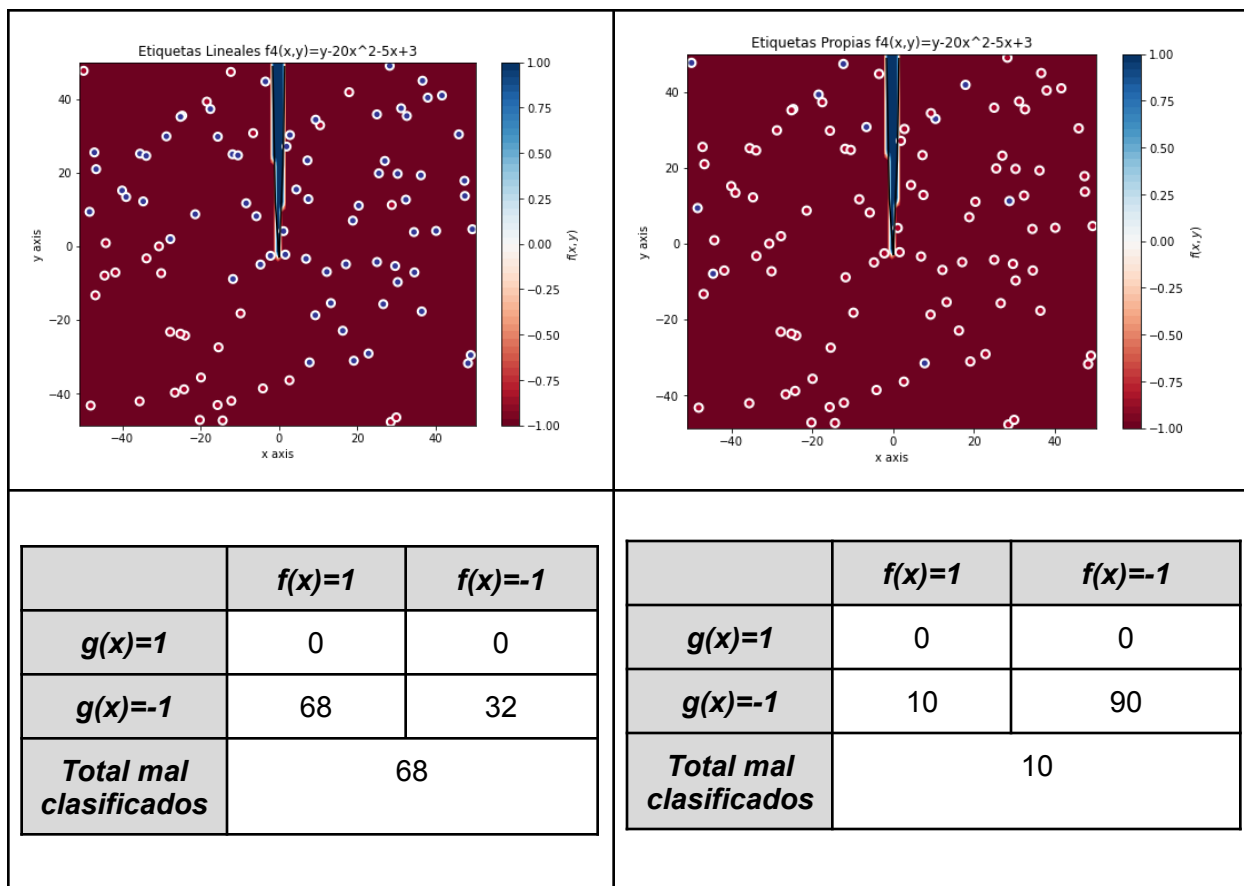
$$f_3(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$$



Podemos observar cómo para la muestra etiquetada por la función lineal clasifica 12 puntos como “1” cuando realmente son “-1”, así como ha clasificado 65 puntos como “-1” cuando realmente son “1”. Y para la muestra etiquetada por ella misma, obtiene un error de un 10% en cada clase, un total de 9 errores

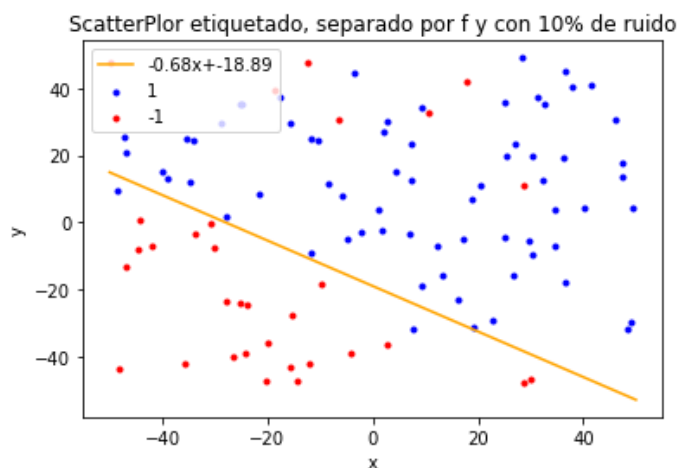
$$f_4(x, y) = y - 20x^2 - 5x + 3$$

Frontera de etiquetado( $y=ax+b$ )	Frontera de etiquetado( $f_4$ )
------------------------------------	---------------------------------



Podemos observar cómo para la muestra etiquetada por la función lineal no ha clasificado ningún “1” mal, el problema es que no ha clasificado nada como “1”, pero en cambio ha clasificado 68 puntos como “-1” cuando realmente eran “1”. Y para la muestra etiquetada por ella misma, obtiene un error de un 10% en cada clase, un total de 10 errores

Si recuperamos la matriz de confusión obtenida por la clasificación generada por recta:



	$f(x)=1$	$f(x)=-1$
$g(x)=1$	66	7
$g(x)=-1$	2	25
<b>Puntos mal clasificados:</b>	9	



Podemos afirmar que la recta es claramente mejor clasificadora que las funciones anteriores **para la muestra etiquetada por la recta**, aunque las otras funciones sean más complejas. Ya que como hemos calculado con las matrices de confusión la recta obtiene 9 errores en una muestra de tamaño 100, pero de las funciones anteriores, que son más complejas, la mejor de ellas ha obtenido 42 errores sobre una muestra de tamaño 100.

También Podemos ver como cuando las funciones **generan sus propias etiquetas y se le aplica un ruido**, este **afecta de la misma forma a todas ellas**, independientemente de que sean una recta o una función mucho más compleja, el ruido afecta a todas por igual, en todas las funciones obtenemos un 10% de etiquetas mal clasificadas.

Por lo que a la conclusión que podemos llegar es que utilizar funciones más complejas no va a reducir el error generado por el ruido. Aun así no hay que tratar de clasificar correctamente el ruido, el mejor clasificador de una población va a ser el que separe ambas clases como si el ruido no estuviera, aunque esto en un problema real es difícil de saber, ya que es difícil diferenciar qué es ruido y que no, no hay forma de modelarlo. Si la muestra no fuese linealmente separable, las funciones anteriores podrían ser muy útiles para reducir el error, pero no el producido por el ruido, si no el propio error generado por el mal ajuste a la frontera de clasificación, como hemos visto cada función de las anteriores es la mejor para clasificar las instancias etiquetadas por ella misma, los cuales son dataset no linealmente separable.

## 2.-Modelos lineales

### 2.1.-Algoritmo del perceptrón

Antes de comentar el algoritmo completo vamos a ver cómo se calcula la función de error de una clasificación y la modificación de los pesos del perceptrón.

A la hora de calcular el error que genera un ajuste de  $w$  en la muestra, lo único que hay que hacer es contar las instancias que están mal clasificadas y dividir entre el número de instancias. Una instancia está mal clasificada si al obtener el signo de la predicción del modelo, este es distinto al de las etiquetas.

$$\text{sign}(w^T x_i) \neq y_i$$

Por lo que ya únicamente haciendo la media de todos los errores obtenemos la siguiente expresión.

$$\text{Error}(w) = \frac{1}{N} \sum_{i=1}^N [\text{sign}(w^T x_i) \neq y_i] = \frac{1}{N} \sum_{i=1}^N \begin{cases} 1 & \text{if } \text{sign}(w^T x_i) \neq y_i \\ 0 & \text{if } \text{sign}(w^T x_i) = y_i \end{cases}$$

```
def Err_PLA(x,y,w):  
    error = 0  
    for caracteristicas, etiqueta in zip(x,y):  
        #Si está mal clasificada  
        if(signo(np.matmul(w,caracteristicas))!=etiqueta):  
            error+=1#Añado uno al error total  
    return error/y.shape[0] #Divido entre el número de instancias
```

El algoritmo del perceptrón se basa en la siguiente modificación de pesos:

Nombre: Alberto Robles Hernández  
DNI: 76065648-W

$$new\ w = \begin{cases} w + y_i x_i & \text{if } sign(w^T x_i) \neq y_i \\ w & \text{if } sign(w^T x_i) = y_i \end{cases}$$

En cada iteración para decidir la modificación de  $w$  sólo se utiliza una instancia, que si está bien clasificada no hace nada y si está mal clasificada se modifica  $w$  para añadir dicha instancia en la clase que corresponda. Es por esto por lo que se dice que es un algoritmo sin memoria, por que no tiene en cuenta otras instancias a la hora de modificar  $w$ , y puede llegar a soluciones peores con tal de añadir la instancia con la que esté trabajando en ese momento en la clase que le corresponda.

El algoritmo del perceptrón lo único que hace es iterar sobre toda la muestra un número de veces determinado por dos factores:

- Que el número de iteraciones sobre toda la muestra sea menor que un número de iteraciones máximo que le pasamos como parámetro
- Que cuando hayamos recorrido toda la muestra no se haya producido ningún cambio en los pesos, o lo que es lo mismo, que todas las instancias estén bien clasificadas

Cuando estemos iterando sobre una instancia de la muestra lo único que tenemos que hacer es ver si dicha instancia está bien clasificada, en cuyo caso no hacemos nada o mal clasificada, que en este caso aplicamos la modificación de los pesos que hemos visto anteriormente.

```
def ajusta_PLA(datos, label, max_iter, vini):  
    #w pasa a ser el valor inicial  
    w=np.copy(vini)  
    modificado = True  
    numIteraciones=0  
    #Mientras algún punto no este bien ajustado y el número de iteraciones  
    #sea menor que el máximo, seguimos iterando  
    while modificado and numIteraciones<max_iter:  
        modificado = False  
        #Iteramos sobre todas las instancias  
        for caracteristicas, etiqueta in zip(datos,label):  
            #Si una instancia está mal clasificada modificamos w  
            if(signo(np.matmul(w,caracteristicas))!=etiqueta):  
                w = w + etiqueta*caracteristicas  
                modificado = True  
        numIteraciones+=1  
    return w, numIteraciones
```

El algoritmo es bastante simple, pero tiene un problema, y es que en caso de que converja y pare por que no se han modificado los pesos tenemos la certeza de que el menos la muestra la clasifica perfectamente, pero en caso de gaste todas las iteraciones, es decir que llegue al máximo de iteraciones podemos tener dos situaciones:

- Que el número de iteraciones no sea suficiente para converger, bastaría con aumentar el número de iteraciones máximas
- Que el dataset no sea linealmente separable, por lo que nunca podrá clasificar la muestra perfectamente y esto hará que itere hasta el número máximo de iteraciones. Pero esto nos crea otro problema y es que como hemos dicho antes es un algoritmo sin memoria, por lo que seguramente hayamos pasado por una solución más o menos

buena, pero como se ha dado cuenta de que no todas las instancias están bien clasificadas ha podido ir a soluciones peores. La solución a esto es el algoritmo del perceptrón pocket. Lo único que hace es guardar la mejor solución que vea durante toda la ejecución del algoritmo, y cuando el algoritmo termina devuelve dicha solución. El resto de aspectos son todos iguales al algoritmo del perceptrón normal.

```
def ajusta_PLA_Pocket(datos, label, max_iter, vini):  
    #w pasa a ser el valor inicial  
    w=np.copy(vini)  
    #Guardamos siempre el mejor valor hasta el momento para que en caso de que  
    #no converge no tener un valor muy malo  
    mejor_w = np.copy(vini)  
    #Calculamos el error del ajuste inicial  
    menor_error = Err_PLA(datos,label,w)  
  
    modificado = True  
    numIteraciones=0  
    #Mientras algún punto no este bien ajustado y el número de iteraciones  
    #sea menor que el máximo, seguimos iterando  
    while modificado and numIteraciones<max_iter:  
        modificado = False  
        #Iteramos sobre todas las instancias  
        for caracteristicas, etiqueta in zip(datos,label):  
            #Si una instancia está mal clasificada modificamos w  
            if(signo(np.matmul(w,caracteristicas))!=etiqueta):  
                w = w + etiqueta*caracteristicas  
                modificado = True  
        error_actual = Err_PLA(datos,label,w)  
        #Si el error actual es menor que el mejor guardamos w actual como mejor  
        if(error_actual<menor_error):  
            menor_error = error_actual  
            mejor_w = np.copy(w)  
        numIteraciones+=1  
    return mejor_w, numIteraciones
```

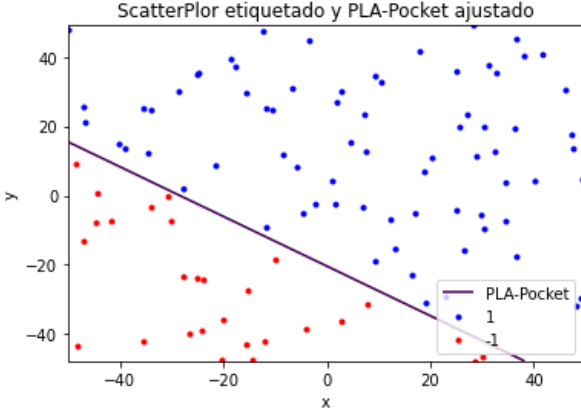
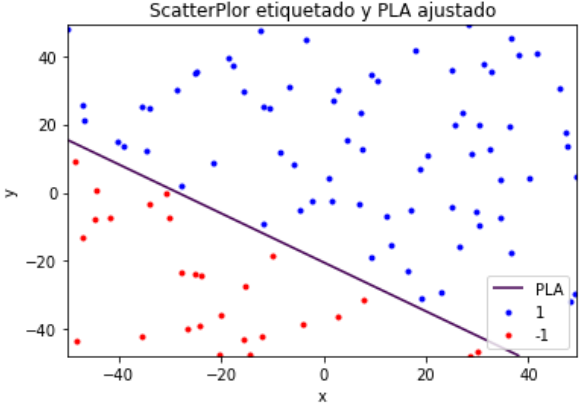
Para los ejercicios he utilizado esta última aproximación perceptrón-pocket

### 2.1.1.-Aplicación de PLA sobre dataset sin ruido

A Continuación he aplicado el algoritmo PLA sobre los datos obtenidos en el ejercicio anterior, sin ruido.

Me he tomado la libertad de hacer dos ejecuciones con punto de inicio [0,0,0], una con PLA y otra con PLA-Pocket, para demostrar lo que he comentado antes acerca de la parada y la separación lineal de la muestra. Podemos observar que hemos obtenido los mismo resultados para ambos algoritmos, esto es por que el dataset es linealmente separable y el algoritmo no tiene ningún problema en ver que ha separado ambas clases perfectamente en la muestra, parar y devolver dicha solución. Luego veremos un ejemplo para ver qué ocurre cuando la muestra no es linealmente separable.

Podemos ver también que el algoritmo tanto PLA y PLA-Pocket es determinista, para un mismo punto inicial, obtenemos la misma solución en el mismo número de pasos.

	PLA-Pocket	PLA
Núm itr	75	75
W	[661. 23.2024 32.3916]	[661. 23.2024 32.3916]
Error	0.0	0.0
Scatter plot		

```
PLA, Valor inicial [0,0,0]
w:[ 661.      23.20241712  32.39163606 ]
Numero iteraciones: 75
Error: 0.0

PLA-Pocket Valor inicial [0,0,0]
w:[ 661.      23.20241712  32.39163606 ]
Numero iteraciones: 75
Error: 0.0
```

Inicializando el valor inicial de forma aleatoria obtenemos las siguientes iteraciones:

	Valor Inicial	Número de iteraciones
1	[0.61851357 0.01036426 0.53862728]	60
2	[0.00301796 0.95119379 0.90540203]	248
3	[0.79596694 0.91527432 0.14555823]	43
4	[0.15773007 0.18763167 0.6224959 ]	72
5	[0.9058095 0.98995518 0.71112246]	129
6	[0.73180041 0.9092932 0.40087373]	244
7	[0.24985068 0.17343017 0.11945705]	70
8	[0.81261059 0.14679237 0.26429748]	84
9	[0.81908918 0.31058725 0.98241745]	122

10	[0.2666387 0.53365334 0.31446701]	37
----	-----------------------------------	----

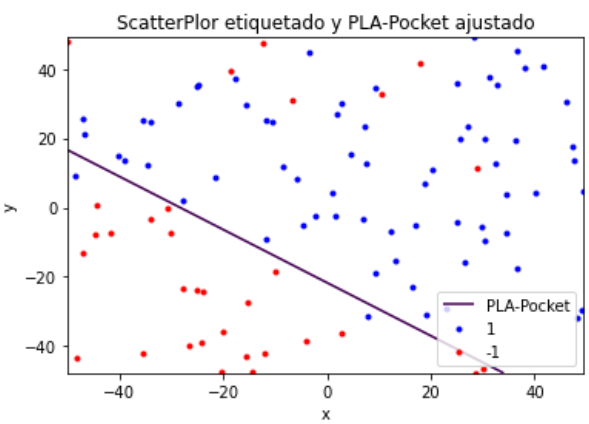
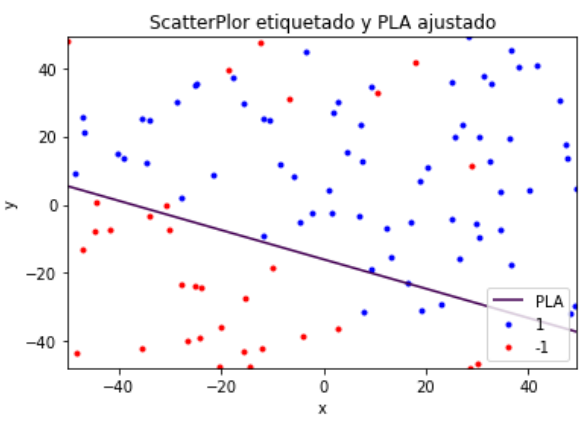
El número medio de iteraciones necesarias para converger es 110.9, como vemos converge en un número de iteraciones “razonable”, no itera indefinidamente, lo que significa que para por que ha encontrado un ajuste donde todos las instancias están bien clasificadas, no por que llegue al número máximo de iteraciones.

Como conclusión podemos decir que el algoritmo PLA es muy sensible al punto de inicio, el número de iteraciones necesarias para converger depende directamente de este este, siempre que la muestra sea linealmente separable. Como podemos ver para cada punto inicial hemos obtenido un número de iteraciones distinto.

### 2.1.2.-Aplicación de PLA sobre dataset con ruido

A Continuación he aplicado el algoritmo PLA sobre los datos con ruido obtenidos en el ejercicio anterior.

Ahora ejecutando de nuevo PLA y PLA-Pocket inicializados en [0,0,0] podemos observar como no obtenemos los mismos valores, ya que esta vez debido al ruido las dos clases no son linealmente separables, por lo los algoritmos no paran hasta no agotar el número máximo de iteraciones. Pero la diferencia es que PLA-Pocket guarda la mejor solución que ha encontrado, y como podemos observar clasifica menos puntos mal, que PLA, y los que clasifica mal es debido al ruido, pero PLA como no guarda la mejor solución devuelve la solución obtenida en la última iteración, la cual puede no ser la mejor.

	PLA-Pocket	PLA
Núm itr	1000	1000
W	[470. 16.5750 21.6439]	[647. 17.3126 40.3828]
Error	0.09	0.11
Scatter plot		

```
PLA, Valor inicial [0,0,0]
w:[ [647.      17.31263139  40.38288826] ]
Numero iteraciones: 1000
Error: 0.11

PLA-Pocket Valor inicial [0,0,0]
w:[ [470.      16.5750085   21.64390456] ]
Numero iteraciones: 1000
Error: 0.09
```

Para la inicialización aleatoria de los puntos iniciales hemos obtenido lo siguiente:

	Valor Inicial	Número de iteraciones
1	[0.91077283 0.36655664 0.43359233]	1000
2	[0.51229269 0.93888648 0.03094901]	1000
3	[0.71687866 0.89101895 0.02728722]	1000
4	[0.52205125 0.32598981 0.85948932]	1000
5	[0.55851655 0.69022787 0.4528535 ]	1000
6	[0.62830904 0.29009685 0.00934858]	1000
7	[0.57675593 0.31144421 0.5172676 ]	1000
8	[0.91640585 0.42647479 0.24739604]	1000
9	[0.37129376 0.93186112 0.93686838]	1000
10	[0.84432995 0.92020651 0.22790029]	1000

El número de iteraciones medio es 1000.

Por la misma explicación que he dado antes, los estados iniciales aleatorios también van a tardar 1000 iteraciones en acabar, ya que da igual donde lo inicializamos, siempre va a modificar algún elemento ya que es imposible separar ambas clases linealmente, e itera hasta el número máximo de iteraciones. Pero si guardáramos de las distintas ejecuciones la iteración en la que encuentran la mejor solución nos encontraríamos con algo muy parecido al dataset linealmente separable, que algunos tardan muy poco en encontrar la mejor solución y sin embargo otros tardan mucho.

## 2.2.-Regresión Logística

Antes de explicar la regresión logística voy a explicar el error sigmoidal y la modificación que utilizaremos para w

El error de un ajuste w sobre una muestra de datos x es la media del error sigmoidal de cada instancia:

$$E_{in}(w) = \frac{1}{N} \sum_{n=1}^N \ln \left( 1 + e^{-y_i w^T x_i} \right)$$

```
def Err_sig(x,y,w):  
    error = 0  
    for características, etiqueta in zip(x,y):  
        #Acumulamos el error sigmoial que generan cada una de las instancias  
        error += np.log(1 + np.e**(-(etiqueta*np.matmul(w.T,características))))  
    return error/y.shape[0]#Dividimos entre el numero de instancias
```

Derivando la expresión anterior obtenemos el gradiente en un punto para un conjunto de instancias:

$$\nabla E_{in} = -\frac{1}{N} \sum_{n=1}^N \frac{y_n x_n}{1 + e^{y_n w^T x_n}}$$

```
def gradErrSig(x,y,w):  
    #Creamos un array de ceros con el mismo tamaño que w  
    newW = np.zeros(w.shape[0])  
    #Iteramos sobre cada elemento de x, y acumulamos en w  
    for i in range(y.shape[0]):  
        newW += (y[i]*x[i])/(1+np.e**(y[i]*np.matmul(w.T,x[i])))  
    return -newW/y.shape[0]#Dividimos entre el número de instancias
```

Por lo que el incremento de w se nos queda de la forma

$$new\ w = w - \eta \nabla E_{in}(w)$$

Lo único que nos queda para tener totalmente definido el algoritmo es saber cuántas instancias queremos utilizar para calcular el descenso del gradiente, este tema ya lo abordamos en la práctica anterior, podríamos utilizar la muestra entera (*descenso del gradiente Bach*) o utilizar subconjuntos de la muestra (*descenso de gradiente estocástico*).

En mi caso he definido el algoritmo como descenso del gradiente estocástico.

Algunos factores a tener en cuenta implícitos en el SGD pero que voy a mencionar, es que tras iterar una vez sobre todos los mini baches hay que barajar todas las instancias y volver a obtener los mini baches.

Como criterio de parada, a parte del número de iteraciones máximo vamos a añadir otro, y es que la norma de la diferencia entre el w resultante de iterar sobre todos los mini baches y modificarlo, con el w que había antes de iterar sea menor que un valor dado, entonces acaba el algoritmo

$$\|w^{(t-1)} - w^{(t)}\| < diferenciaMinima$$

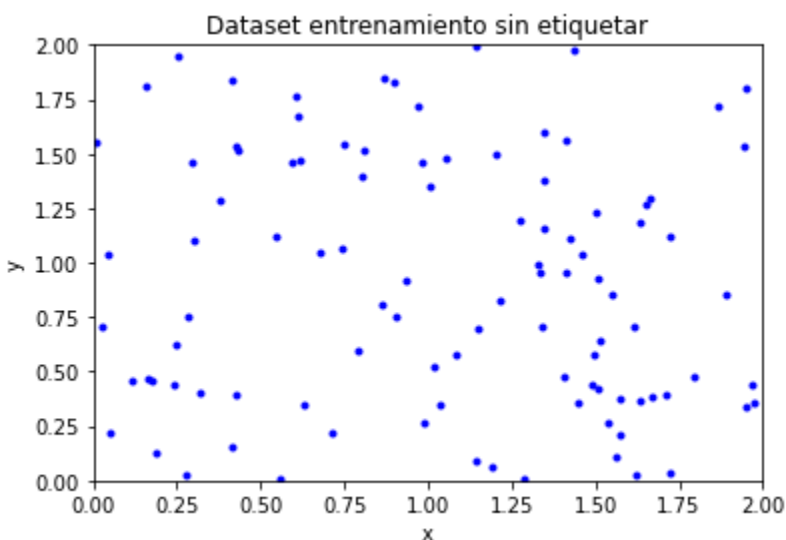
```
def sgdRL(x, y, w, numMaxIter, learning_rate, diferenciaMinima=0.01, tamanoBachs=1):  
    iterations=0  
    #Asignamos a diferencia actual un valor mayor que diferencia mínima para  
    #entrar en el bucle  
    diferenciaActual = diferenciaMinima+1  
    #Mientras el numero de iteraciones sea menor que el máximo y la norma  
    #de la diferencia entre w(t)-w(t+1) sea menor que diferenciaMinima  
    while(iterations<numMaxIter and diferenciaMinima<diferenciaActual):  
        #Generamos un vector que indexe cada una de las instancias
```

```
indices = np.arange(y.shape[0])
#Barajamos los índices
np.random.shuffle(indices)
#Obtenemos una lista de mini-Batches de la muestra
mis_batches = list(getBachs(indices,tamanoBachs))
#Guardamos el ajuste actual para compararlo posteriormente
w_old = np.copy(w)
for mi_batch in mis_batches:
    #Calculamos el incremento a W del descenso de gradiente sigmoidal
    incrementoW = - learning_rate * gradErrSig(x[mi_batch:],y[mi_batch:],w)
    #Modificamos w
    w = w + incrementoW
iterations+=1
#Calculamos la norma de la diferencia de ajustes actual |w_old - w|
diferenciaActual=np.linalg.norm(w_old-w)

return w, iterations
```

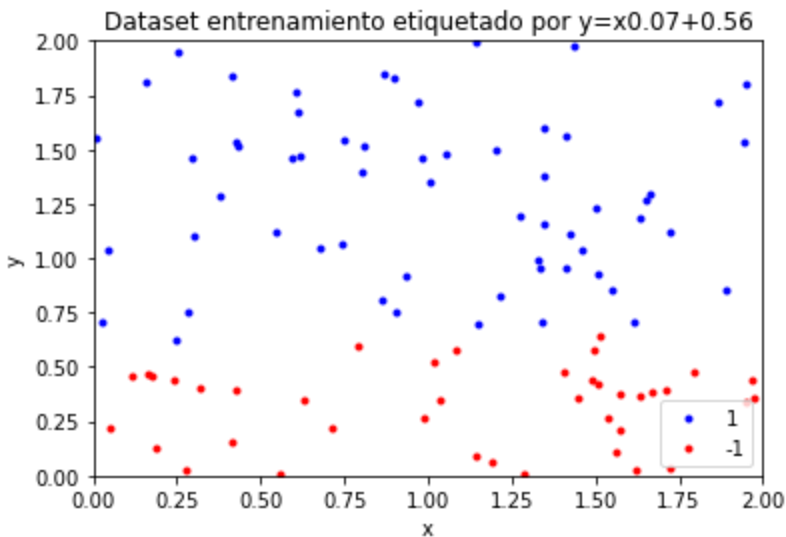
### 2.2.1.-Aplicación de SGD-RL sobre dataset

Vamos a aplicar la regresión logística mediante el descenso de gradiente estocástico sobre una muestra de 100 instancias creada con la función `simula_unif(N, dim, rango)`, generada con un distribución uniforme, en 2 dimensiones, en el rango [0,2]

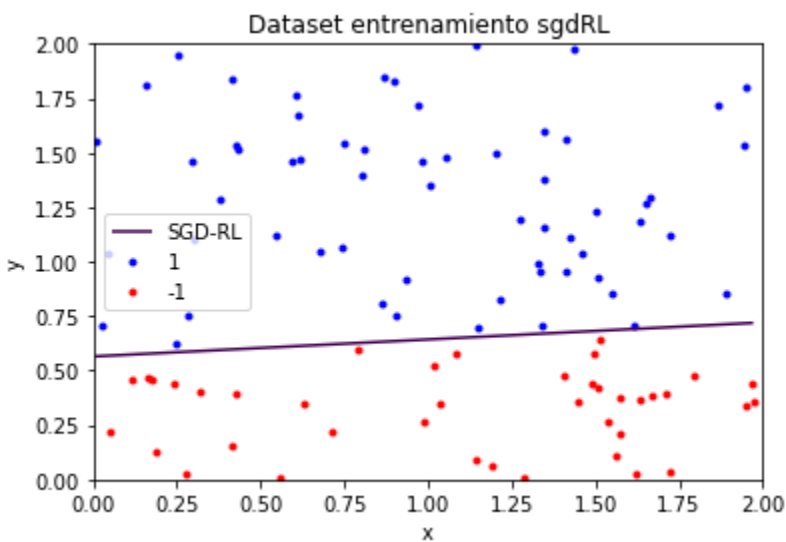


Separando las instancias en dos clases por una recta aleatoria generada con la función `simula_recta(rango)` y etiquetando los puntos que estén a un lado de la recta como 1 y los que estén al otro como -1 obtenemos el siguiente etiquetado





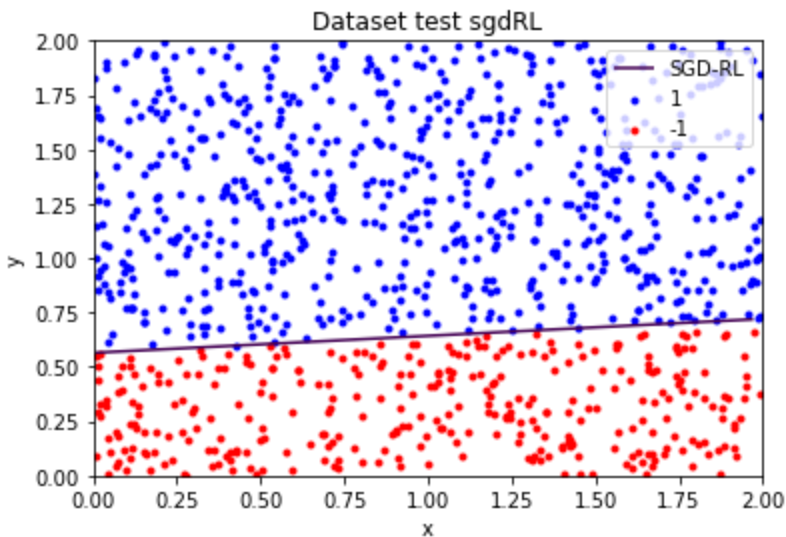
Aplicando el SGD-RL sobre la muestra anterior, con un learning rate de 0.01, una norma de la diferencia entre ajustes mínima de 0.01 y el ajuste inicial = [0,0,0] obtenemos el siguiente ajuste:



Obtenemos un valor de error sigmoidal en la muestra: 0.0932588806525.

Que el error sigmoidal no sea 0, no quiere decir que estamos clasificando erróneamente algunos puntos, simplemente es que la probabilidad de ciertos puntos de pertenecer a su clase es ligeramente menor que 1.

Ahora vamos a generar un conjunto de datos para test, que consiste en generar 1000 puntos de la misma forma que antes, pero esta vez no volvemos a generar una recta aleatoria, utilizamos la que habíamos generado anteriormente y etiquetamos la muestra.



Obtenemos un valor de error sigmoidal en el test: 0.093764622276

Podemos ver que el error sigmoidal en la muestra es ligeramente menor que en el test, ambos son errores bastante bajos, hemos encontrado un buen ajuste.

### 2.2.2.-Experimento

Para realizar el experimento simplemente realizamos el apartado anterior 100 veces

- Generamos 100 puntos y los etiquetamos en función de su posición respecto a una recta que hemos generado anteriormente
- Ajustamos una función a la muestra mediante SGD-RL
- Generamos 1000 puntos y los clasificamos mediante la recta aleatoria obtenida anteriormente
- Calculamos el error de nuestro modelo en la muestra anterior

Calculando la media de los errores que se producen en el conjunto de datos de test, así como el número medio de iteraciones que tarda el algoritmo SGD-RL en converger

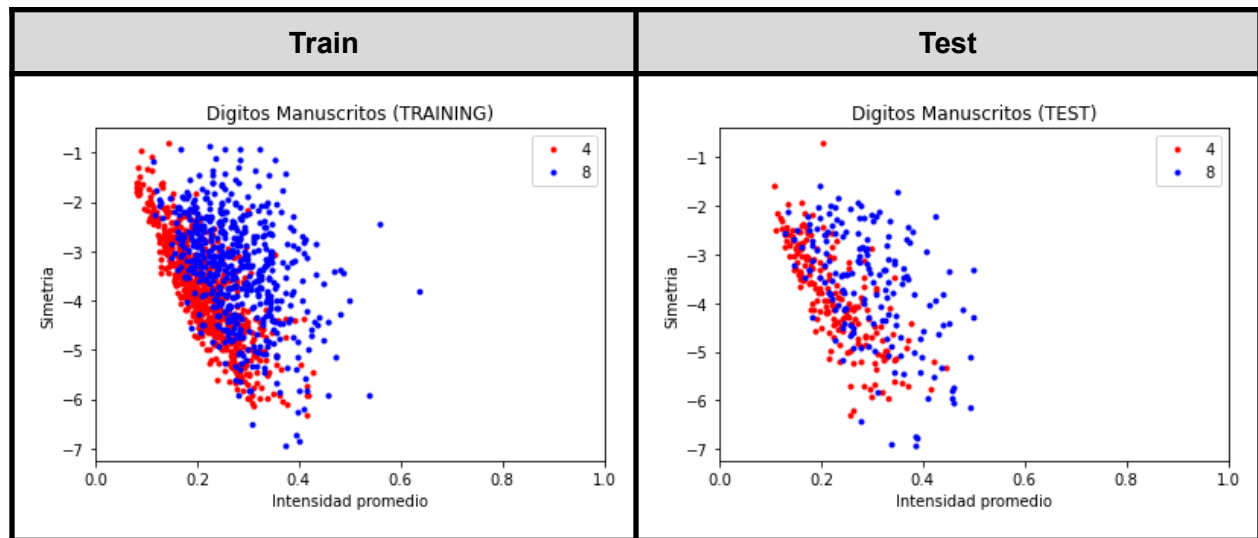
Número medio de iteraciones	439.49
Error sigmoidal medio en Test	0.11984889784490027

El error medio en test aumenta ligeramente respecto al error de test calculado anteriormente para una sola muestra.

## 3.-Bonus

Vamos a intentar clasificar dígitos manuscritos entre 4 y 8.

Como características disponemos de su simetría e intensidad promedio.



### 3.1.-Clasificando dígitos con Pseudo-Inversa y mejorando con PLA-pocket.

Como es un problema de clasificación binario y cada clase está etiquetada como 1 o -1 podemos utilizar la regresión para resolver un problema de clasificación. He elegido la pseudo-inversa para calcular el primer ajuste, el cual le pasaremos después como punto inicial al PLA-pocket.

#### 3.1.1.-Pseudo-Inversa

Recordando un poco como se ajusta un modelo mediante la pseudo inversa, lo único que hay que hacer es obtener evidentemente la pseudo inversa de las características o matriz X.

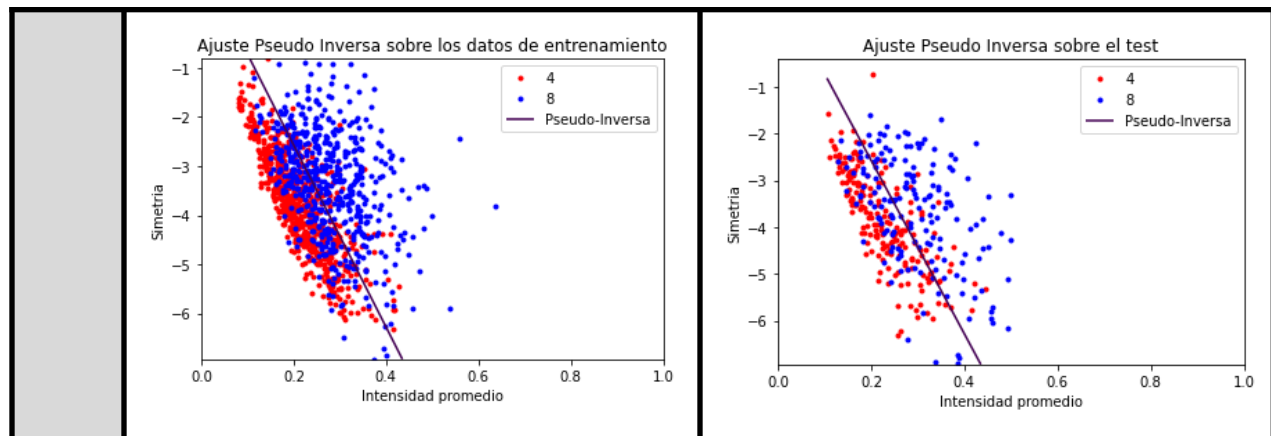
$$X^{\dagger} = (X^T X)^{-1} X^T$$

Una vez tenemos la pseudo inversa para obtener los pesos(w) que ajustan el modelo simplemente tenemos que hacer la siguiente operación matricial:

$$W = X^{\dagger} Y$$

Aplicando la regresión lineal mediante la pseudo inversa para clasificar dos clases, obtenemos lo siguiente:

	Training	Test
Error	0.22780569514237856	0.25136612021857924



### 3.1.2-PLA-pocket

Aplicando PLA-Pocket con los siguientes valores

- Punto inicial = Solución obtenida de la pseudo-inversa
- Número máximo de iteraciones= 5000

Obtenemos lo siguiente:

	Training	Test
Error	0.22529313232830822	0.2540983606557377

Podemos observar cómo hemos obtenido una leve mejora en el error en la muestra aplicando el PLA-Pocket respecto a la pseudo-inversa, aunque en el conjunto de datos de test el algoritmo PLA-Pocket empeora el error levemente respecto al error que obtenemos en el test con la Pseudo-inversa

### 3.2.-Cotas de $E_{out}$

Vamos a comparar las cotas basadas en  $E_{in}$  y  $E_{test}$

#### 3.2.1.-Cota basada en $E_{in}$

Vamos a aplicar para la cota basada en  $E_{in}$  la cota de Vapnik-Chervonenkis ya que la cota obtenida mediante la desigualdad de Hoeffding no nos valdría, esta solo es válida cuando eliges una función, no para calcular la cota sobre una clase de funciones. Podríamos utilizar

también el proceso de acotación por convergencia uniforme, pero con esto obtenemos una cota muy laxa, por lo que la mejor opción para calcular la cota sobre una clase de funciones  $H$  es la cota de Vapnik-Chervonenkis.

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} \log \frac{4((2N)^{dvc} + 1)}{\delta}}$$

```
def cotaVCsobreClasePerceptron2D(error_in,dvc,N,delta):
    return error_in+np.sqrt((8/N)*np.log(4*((2*N)**dvc+1)/delta))
```

Para calcular la cota de Vapnik-Chervonenkis antes debemos calcular la función de crecimiento de la clase de funciones del perceptrón en dos dimensiones

$$m_H(1) = 2 = 2^1$$

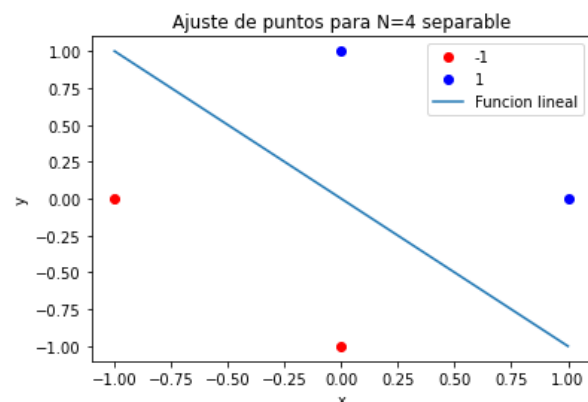
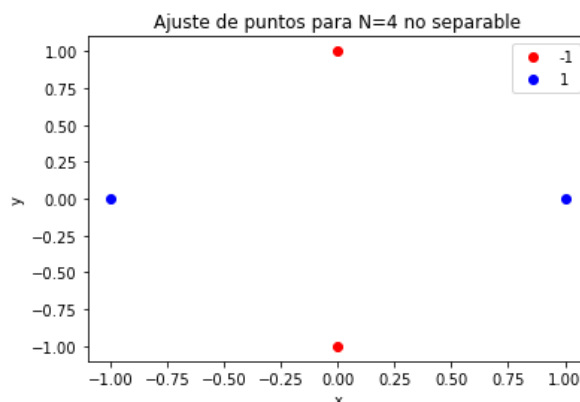
$$m_H(2) = 4 = 2^2$$

$$m_H(3) = 8 = 2^3$$

$$m_H(4) = 14 < 2^4$$

Como podemos observar hemos encontrado el punto de ruptura en  $N=4$ , esto quiere decir que la clase de funciones  $H$  (perceptrón en dos dimensiones), no puede explicar o clasificar perfectamente todas las combinaciones muestrales de 4 instancias.

Por ejemplo, como podemos ver a continuación a la izquierda el perceptrón de 2 dimensiones no es capaz de separar esa distribución de 4 puntos, pero a la derecha vemos como sí puede para otra distribución. Contando el número de distribuciones que puede separar obtenemos  $m_H(4)$  el cual será  $< 2^4$ . Pero para  $N=3$  da igual la clase de cada punto va a poder clasificarlas bien siempre por lo que  $m_H(3)=2^3$



La dimensión de Vapnik-Chervonenkis es el mayor valor de  $N$  que la clase  $H$  puede explicar o clasificar perfectamente, es decir, el punto de ruptura -1.

Por lo que para ya tenemos todos los valores

$$dvc = 3$$

$$\delta = 0.05$$

$$N = 1194$$

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{1194} \log \frac{4((2 * 1194)^3 + 1)}{0.05}}$$

Como tenemos el siguiente error en la muestra

$$E_{in} = 0.2252931$$

Obtenemos la siguiente cota:

$$E_{out}(h) \leq 0.2252931 + \sqrt{\frac{8}{1194} \log \frac{4((2 * 1194)^3 + 1)}{0.05}}$$

$$E_{out}(h) \leq 0.6562296$$

Cota VC basada en  $E_{in}$   
 $E_{out} \leq 0.6562296377990118$

### 3.2.2.-Cota basada en $E_{test}$

Como ahora si tenemos una función preseleccionada, la resultante del ajuste mediante PLA, que no ha sido entrenada con los datos de test, es decir  $|H|=1$ , podemos aplicar la desigualdad de Hoeffding para hallar la cota de  $E_{out}$

$$E_{out}(h) \leq E_{test}(h) + \sqrt{\frac{1}{2N} \log \frac{2}{\delta}}$$

```
def cotaHoeffding(error_test,N,delta):  
    return error_test+np.sqrt((1/(2*N))*np.log(2/delta))
```

Con los siguientes parámetros:

$$\delta = 0.05$$

$$N = 366$$

$$E_{test}(h) = 0.25409836$$

Por lo que obtenemos la siguiente cota:

$$E_{out}(h) \leq 0.25409836 + \sqrt{\frac{1}{2 * 366} \log \frac{2}{0.05}}$$

$$E_{out}(h) \leq 0.32508746$$

Cota Hoeffding basada en  $E_{test}$   
 $E_{out} \leq 0.3250874640883532$

Podemos ver que la cota de  $E_{out}$  basada en  $E_{test}$  (0.3250874640) es 2 veces mejor que la cota que basada en  $E_{in}$  (0.6562296377)

El error en la población es obviamente mayor que el error en test, aunque si aumentamos el número de instancias de test,  $E_{test} \approx E_{out}$ .

Podríamos decir que hemos obtenido un modelo más o menos aceptable, que clasificaría a la población como muy mal con un error de 0.325087.