Práctica 2

Aprendizaje Automático

Blanca Cano Camarero

May 1, 2021

Índice

Ejercicio sobre la complejidad de H y el ruido. **1**
1 Dibujo de las gráficas 1
Influencia del ruido en la clase de funciones. 3
Función de muestra de gráficas 3
c) Nuevas funciones frontera. 7
$f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$ 7
$f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$ 8
$f(x, y) = 0.5(x - 10)^2 - (y - 20)^2 - 400$ 10
$f(x, y) = y - 20x^2 - 5x + 3$ 11
Conclusiones del experimento 13

2 Modelos lineales **15**
Descripción del algoritmo de aprendizaje del perceptrón 15
Apartado 2.a.1.a Ejecución PLA con vector inicial cero 16
Apartado 2.a.1.b Inicialización con vectores aleatorios 18
Apartado 2.a.2 Ejecución para datos con ruido 18
Regresión logística 20
Experimento regresión logística 20

Bonus: Clasificación de dígitos **25**
PLA-pocket 25
Problema a afrontar 25
1. Planteamiento del problema de clasificación binaria. 25
Experimento 27
Experimento 37
Obtención de cotas 38

Ejercicio sobre la complejidad de H y el ruido.

En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir la clase de funciones más adecuada. Haremos uso de tres funciones:

- `simula_unif (N, dim, rango)`, que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo rango.

- `simula_gaus(N, dim, sigma)`, que calcula una lista de longitud N de vectores de dimensión dim, donde cada posición del vector contiene un número aleatorio extraído de una distribución Gaussiana de media 0 y varianza dada, para cada dimensión, por la posición del vector sigma.

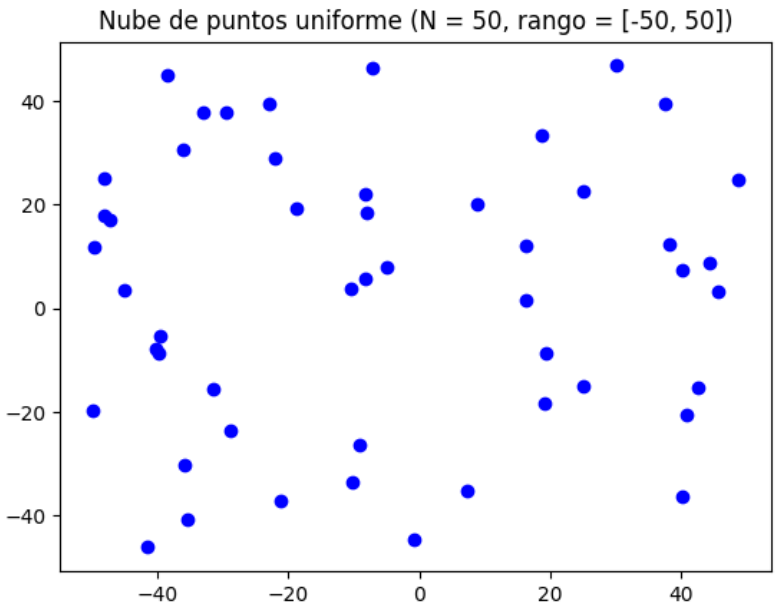
- `simula_recta(intervalo)` , que simula de forma aleatoria los parámetros, v = (a, b) de una recta, $y = ax + b$, que corta al cuadrado $[-50, 50] \times [-50, 50]$.

1 Dibujo de las gráficas

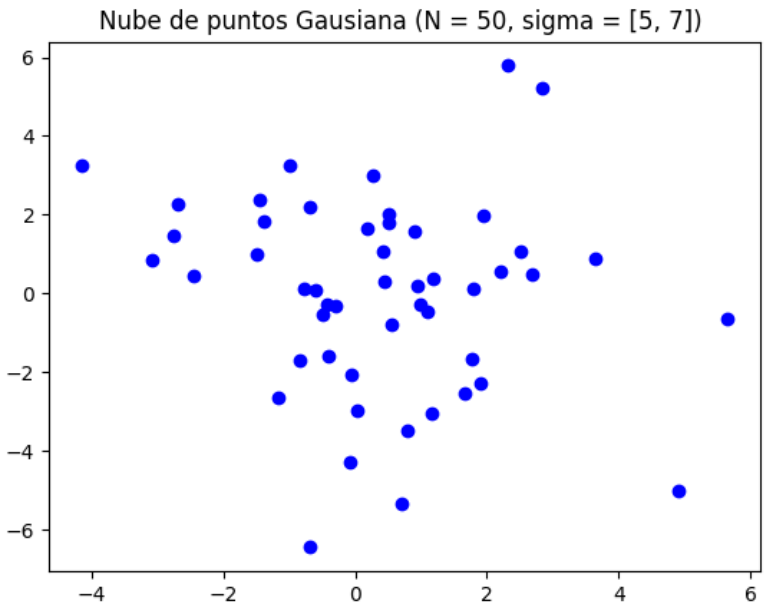
Dibujar gráficas con las nubes de puntos simuladas con las siguientes condiciones:

Para ellos hemos utilizado la función `scatter_plot` que no hace más que agrupar las funciones de visualización.

a) Considere $N = 50, \quad dim = 2, rango = [-50, +50]$ con `simula_unif(N,dim, rango)`.



b) Considere $N = 50$, $dim = 2, sigma = [5, 7]$ con `simula_gaus(N,dim, sigma)`.



Visualmente ambos imágenes son coherentes con la noción de distribución uniforme, las datos se distribuyen aleatoriamente de manera homogénea y en la distribución de Gauss o normal, los valores centrales son más comunes.

Influencia del ruido en la clase de funciones.

Valoración de la influencia del ruido en la selección de la complejidad de la clase de funciones.

Con ayuda de la función `simula_unif(100, 2, [-50, 50])` generamos una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función $f(x, y) = y - ax - b$, es decir el signo de la distancia de cada punto a la recta simulada con `simula_recta()`.

Función de muestra de gráficas

La cabezera de la función con la que dibujaremos las graficas de puntos clasificados y la frontera de función de clasificiación es:

```
def classified_scatter_plot(x,y, function, plot_title, labels, colors):
        '''
        Dibuja los datos x con sus respectivas etiquetas
        Dibuja la función: function
```

```
    y: son las etiquetas posibles que se colorearán
    labels: Nombre con el que aparecerán las etiquetas
    colors: colores de las diferentes etiquetas

    Todo lo dibuja en un gráfico
    '''
```

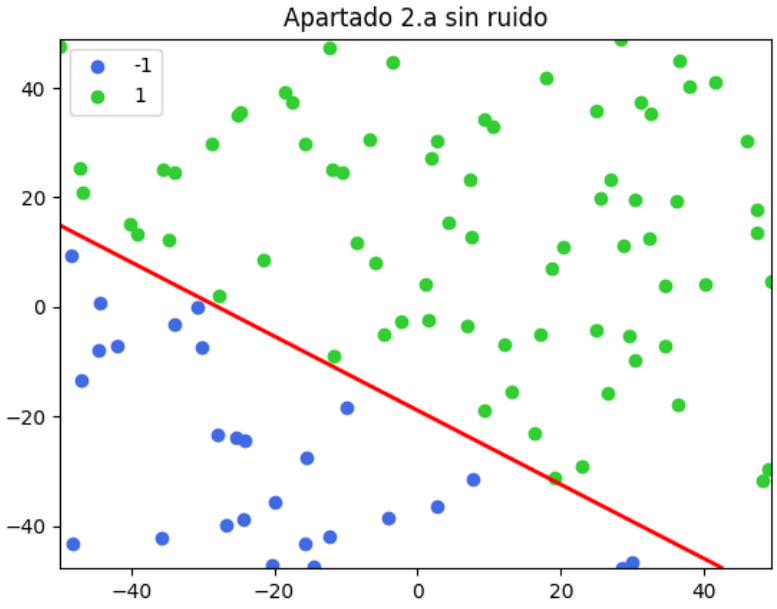
Está implementadas con las funciones `scattered` y `contour` de la librería `matplotlib.pyplot`.

Como único comentario, el límite de división de la función de clasificación $f(x, y)$ de este ejercicio es muy fácil de dibujar de manera manual.

Sabemos que $f(x, y) = 0$ es una recta; luego solo habría que calcular dos puntos de ésta (por ejemplo hacer $x_1 = 0$ e $y_2 = 0$ y resolver las respectivas ecuaciones obteniendo así y_1 y x_2) y pintar la recta que pasa por esos dos puntos (ya lo hicimos en la práctica primera).

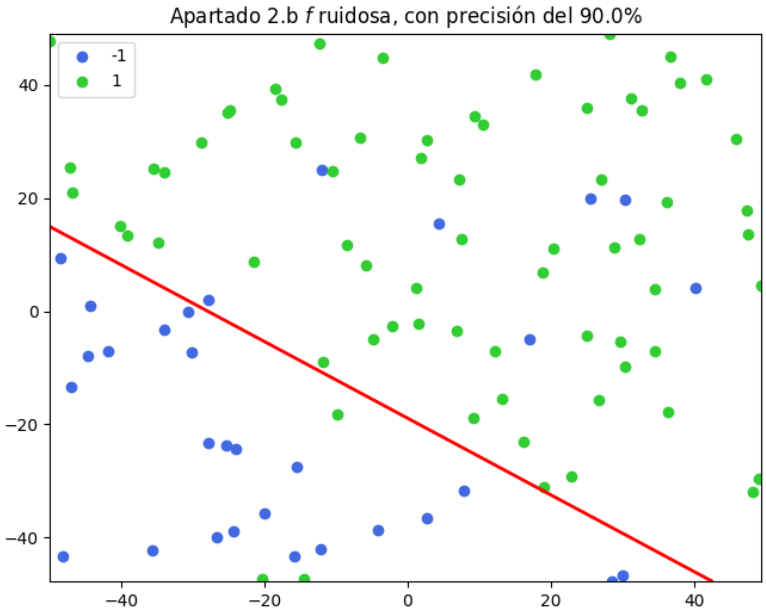
Sin embargo se ha optado por hacer uso de la función `contour` para tener mayor generalidad, ya que esta es capaz de pintar los puntos de cualquier ecuación $g(x, y) = 0$ con $g : \mathbb{R}^2 \longrightarrow \mathbb{R}$ cualquiera.

Etiquetado sin ruido.

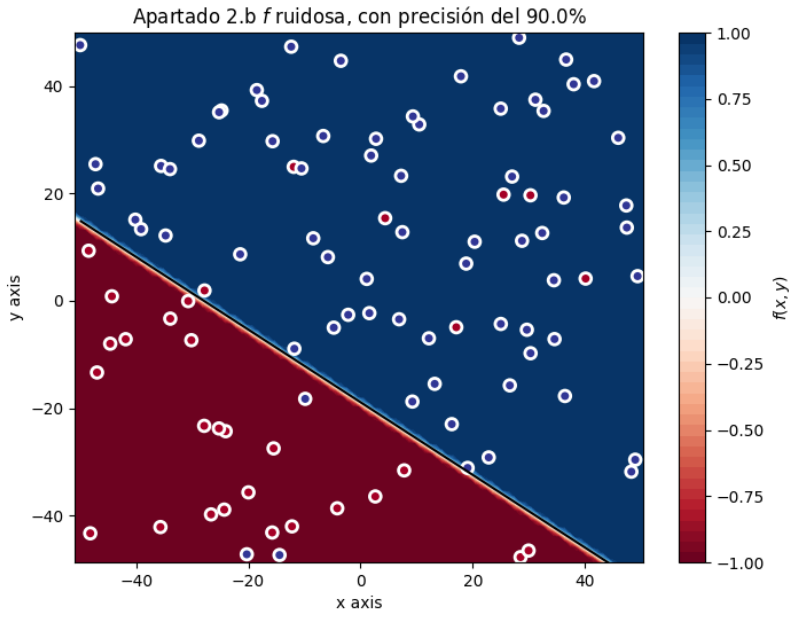


Como podemos ver todo está bien clasificado.

El resultado tras meter ruido es:



También podemos visualizarlo con la función `plot_datos_cuad` porporcionada en el template, que además proporciona información sobre el área de clasificación (de ahora en adelante mostraré las dos gráficas, ya que en mi opinión al usar solo `plot_datos_cuad` no se aprecia tan bien la etiqueta original de los datos).



Podríamos sorprendernos de que solo tres datos de los clasificados como negativos sean ahora positivos, esto se debe a que son menos que los positivos. Analicemos en total el porcentaje cambiado con la función `analis_clasificado` para cerciorarnos de que es correcto

Su salida en ejecución (redondeando a tres decimales los resultado originales) es :

```
Apartado 1.2.b
Resultado clasificación:
     Positivos fallados 7.0  de 73, lo que hace un porcentaje de 9.589
     Negativos fallados 3.0  de 27, lo que hace un porcentaje de 11.111
     Total fallados 10.0  de 100, lo que hace un porcentaje de 10.0
     La precisión es   de 90.0 %
```

Se nos pedía clasificar mal el 10% de los positivos, que son 73, luego eso supondría modificar 7.3 datos mal, puesto que se redondea, la clasificación actual es de 9.58%. Para el caso de los negativos se procede igual.

Sin embargo el resultado final sí que es 10%, lo cual nos termina por confirmar la corrección del algoritmo ya que el error de clasificación sería $malClasificados = 0.1positivos + 0.1negativos = 0.1(positivos + negativos)$ y aunque se ha redondea, como uno ha sido a la alta y el otro a la baja esto hace que se compense el total y el porcentaje final de fallados sea el pedido para subcategoría.

c) Nuevas funciones frontera.

Analizaremos ahora los resultados modificando las funciones frontera:

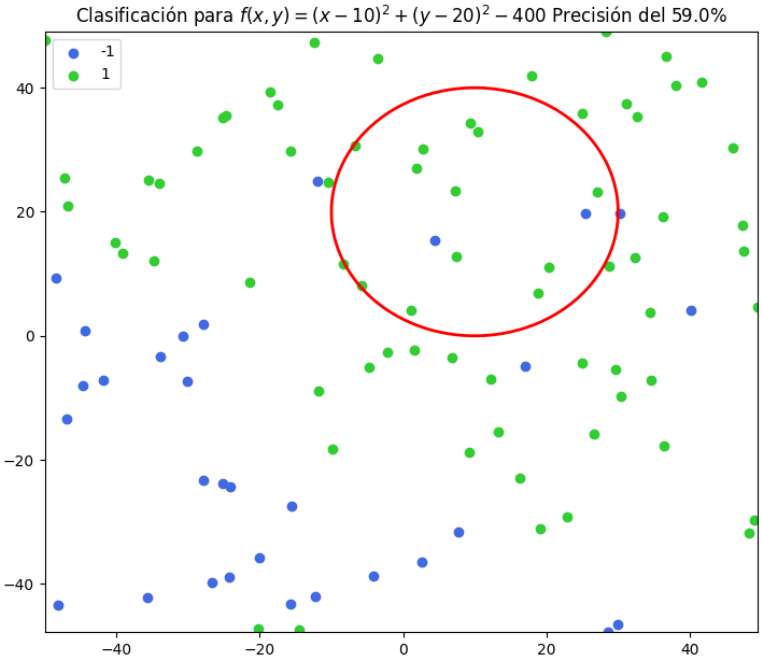
Para analizar la bondad del ajuste vamos a tener en cuenta la precisión, además para comprobar si beneficia más a un tipo u a otro analizando los positivos y negativos fallados.

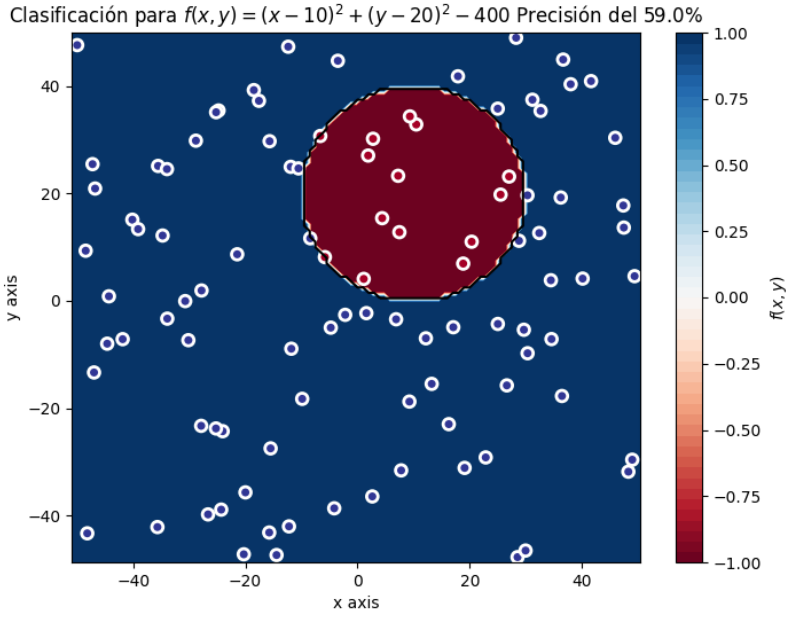
Recordamos que la precisión se define como $\frac{\text{datos bien clasificados}}{\text{datos bien clasificados}}$, nosotros además la indicamos como porcentaje, multiplicada por 100.

Tengamos presente que con la recta hemos obtenido una precisión del 90% y el porcentaje de positivos y negativos fallados era de 10% para ambos.

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

- Precisión obtenida: 59%
- Porcentaje positivos fallados: 17.391%
- Porcentaje negativos fallados: 93.548%

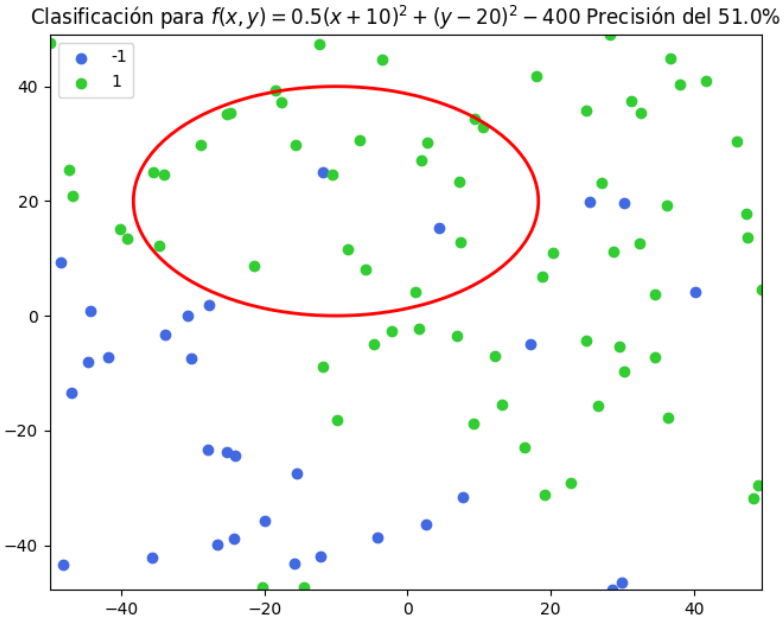


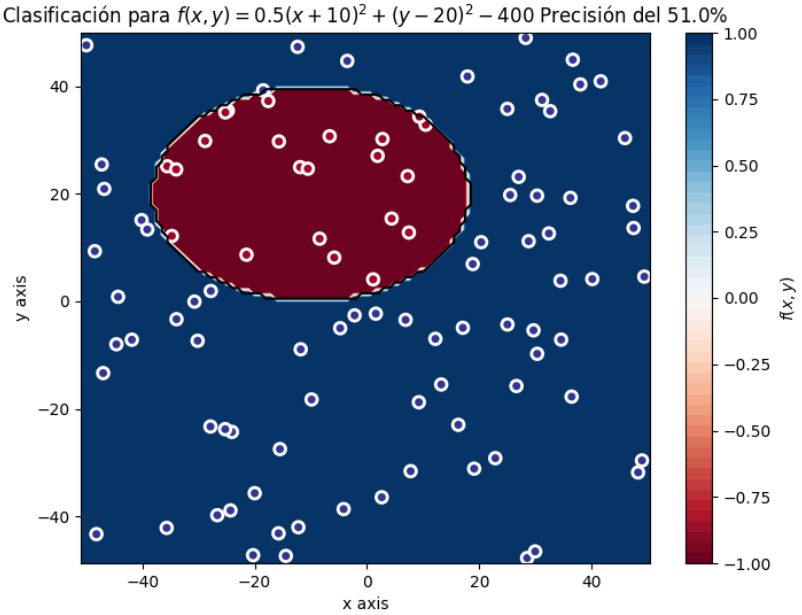


Este caso empeora la precisión y vemos que falla la mayoría de los negativos.

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

- Precisión obtenida: 51.0%
- Porcentaje positivos fallados: 28.986%
- Porcentaje negativos fallados: 93.548%

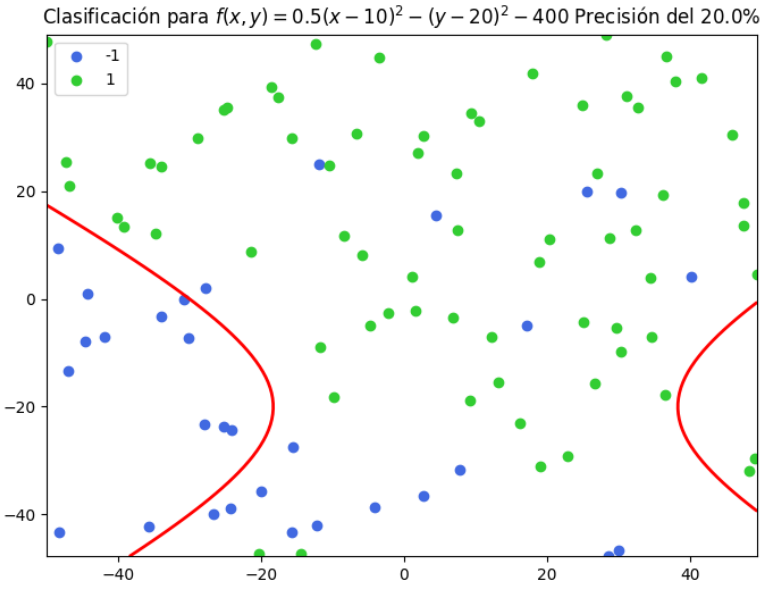


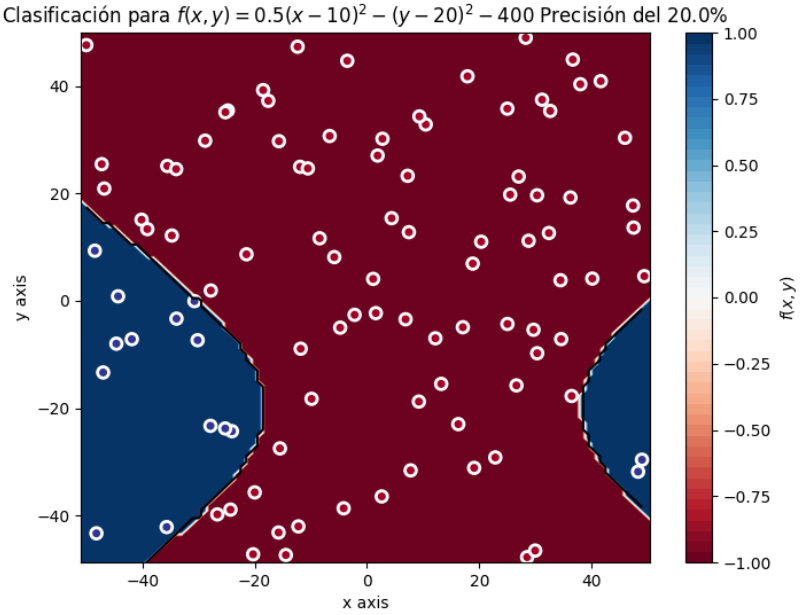


Parecido al ajuste anterior, empeorando la clasificación.

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

- Precisión obtenida: 20%
- Porcentaje positivos fallados: 97.101%
- Porcentaje negativos fallados: 41.935%

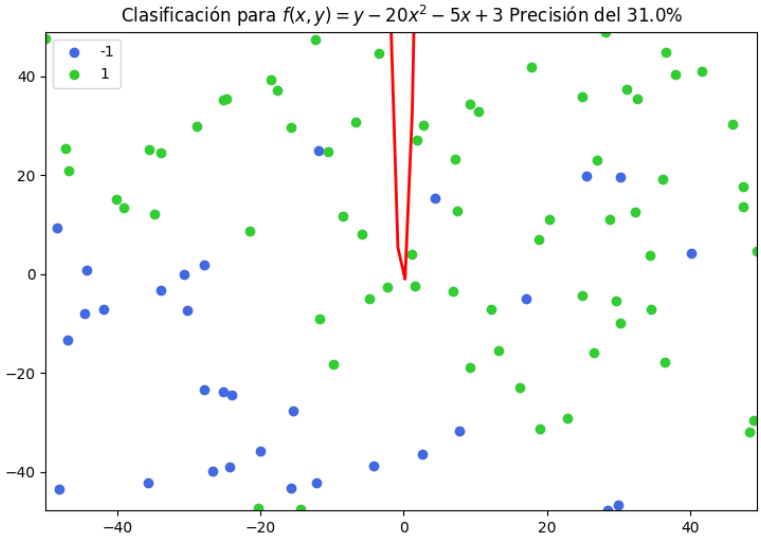


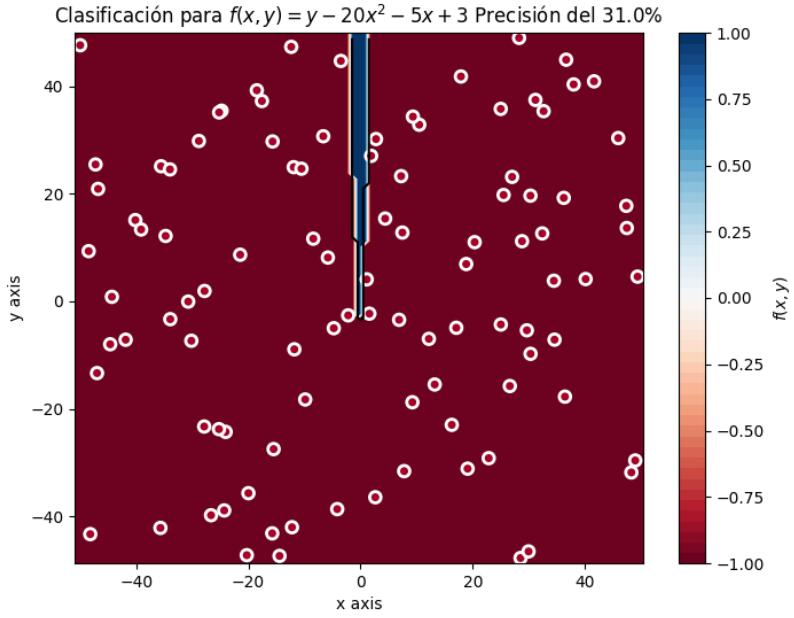


En este caso falla la mayoría de los negativos y es la peor de los ajustes, curiosamente, por la forma que tiene si hubiéramos clasificado con la opuesta $-f(x,y)$, los resultados hubieran sido mucho mejor, un 80% de precisión, lejos aún del 90% inicial.

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

- Precisión obtenida: 31%
- Porcentaje positivos fallados: 100%
- Porcentaje negativos fallados: 0%





Este es un dato curioso, porque directamente lo clasifica todo como positivo, luego la precisión obtenida es la de los positivos que consigue clasificar bien, de ahí que no haya fallado nada clasificando negativos.

Conclusiones del experimento

La clasificación de datos está íntimamente ligada a una función objetivo desconocidad, aunque para clasificar introduzcamos funciones más complejas estas no tienen porqué ajustar mejor que otras más simples y más en este caso donde ni siquiera se están ajustando los coeficientes de estas funciones si no que se nos dan.

Algo muy interesante y que se dejó entrever en la práctica anterior es que aunque utilicimos funciones más complejas que la objetivo desconocida, al ajustar los coeficientes, si el algoritmo verdaderamente converge a la solución; los coeficientes *que aporten complejidad* acabarán por anularse.

2 Modelos lineales

Descripción del algoritmo de aprendizaje del perceptrón

Este algoritmo determinará un vector de pesos $w \in \mathbb{R}^d$ ajustado a través de los como sigue:

El número de características es $d - 1 > 0$.
Sea S un conjunto de pares (x_i, y_i) con $i < |S|$ natural y denotando por $|S|$ el tamaño de muestra, x_i vector de características y y_i etiqueta resultado de la clasificación.

El pseudo código para clasificarlo es:

```
hay_cambio = True
w inicializada

mientras hay_cambio:

    hay_cambio = False

    Para todo (x_i, y_i) en S:
        si signo(w^T x_i) != yi:
            w = w + y_i x_i
            hay_cambio = True

devolver w
```

Si los datos son separables este algoritmo nos asegura la convergencia.

Sin embargo saber con certeza que un conjunto de datos es separable es una hipótesis bastante fuerte y por ejemplo, en el caso de $\mathbb{R}^d$ sabemos que existen configuraciones de $d + 1$ puntos que ya no son clasificables.

Es por ello que para nuestro problema hemos añadido además otro criterio de parada: cuando alcance un número de iteraciones máximo.

Esta solución introduce dos nuevos problemas:

1. Que el conjunto sea separable y converja pero se pare antes de alcanzar dicha solución.

2. Que el w final no sea el mejor de todos los que hemos calculado.

Para resolverlos, se podría plantear una solución en la que se tenga una función para medir el error de cierto w (por ejemplo la contadora de número de datos más clasificados o la precisión), una variable para guardar el valor del w de menor error encontrado y una condición de parada nueva que combine la monotonía del error y el número de iteraciones.

Un ejemplo de algoritmo que mejora esto es el `PLA_pocket` del que hablaremos más adelante en el ejercicios extra.

Apartado 2.a.1.a Ejecución PLA con vector inicial cero

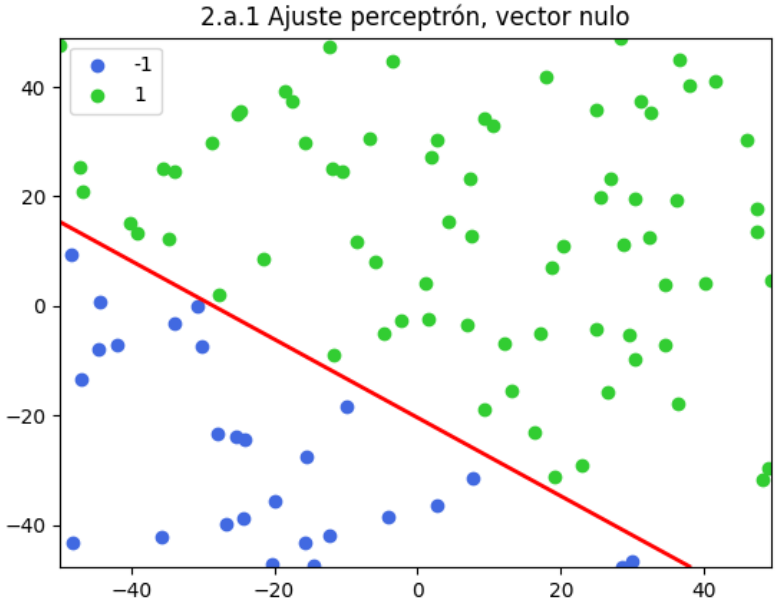
Los coeficientes del vector final son $[661, 23.202, 32.391]$, se ha encontrado tras 75 épocas, antes del máximo de pasos establecido. Esto es gracias a que los datos eran separables y se ha encontrado una solución.

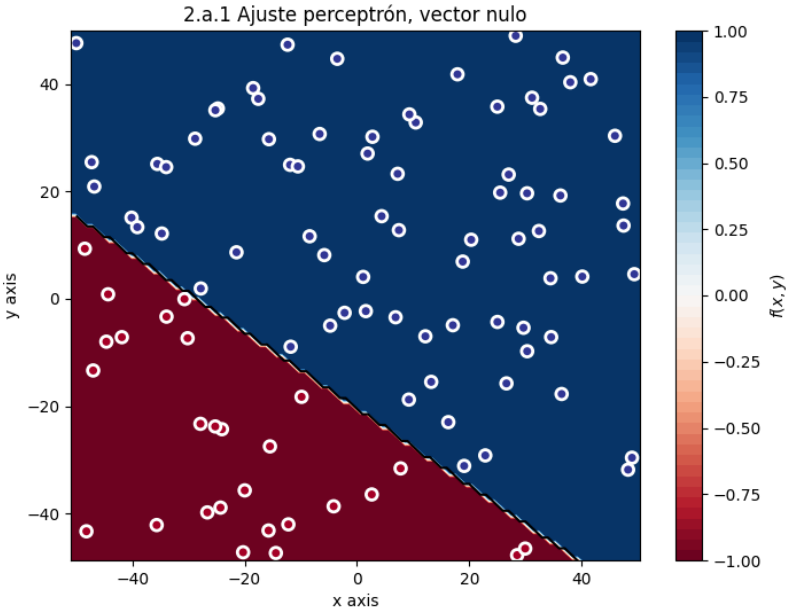
La solución no tiene porqué ser única, de hecho los coeficientes de la recta objetivo original eran $y = ax + b$ con $ a =-0.677, b = -18.89$

Y para conocer los nuestros bastará con despejarlos del vector de pesos recordando que la función de ajuste es el signo de $[1, x, y]\dot{[}w_1, w_2, w_3]^T$ .Luego despejando del producto escalar cundo se anula queda $w_1 + xw_2 + yw_3 = 0$ y dividiendo entre la úlima coordenada queda $y = \frac{w_2}{w_3}x + \frac{w_1}{w_3}$

Por lo que en nuestro caso hemos obtenido $a = \frac{w_2}{w_3} = -0.716$ y $b = \frac{w_1}{w_3} = -20.507$.

En las gráficas correspondientes vemos que efectivamente se ha hecho bien la separación.


2.a.1 Ajuste perceptrón, vector nulo
-1
1
40
20
0
−20
−40
−40
−20
0
20
40


2.a.1 Ajuste perceptrón, vector nulo
y axis
x axis
40
20
0
−20
−40
−40
−20
0
20
40
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
f(x, y)

Apartado 2.a.1.b Inicialización con vectores aleatorios

Se ha utilizado un máximo de 500 iteraciones como heurística a ver que ninguno las incumple, tras 10 iteraciones el número de iteraciones obtenido en cada uno de ellas es: 257, 43, 231, 71, 76, 59, 274, 235, 257, 74.

El número de iteraciones medio es de 157.7 y una desviación típica de 94.175, de aquí deducimos que que nuestro vector inicial sea el nulo es una buena heurística.

Además podemos observar que la desviación típica es bastante grande en comparación con los datos que tenemos, esto nos hace pensar que en el valor inicial tiene relevancia a la hora del número de pasos necesarios.

Analicemos con más detalle el experimento,

En esta tabla se han recogido los datos por número de pasos necesarios,

w_0 es el vector de pesos inicial y w_f el final

numero pasos	w_0	w_f
43	[0.228 0.664 0.497]	[464.228 15.388 23.746]
59	[0.032 0.093 0.065]	[558.032 19.363 29.714]
71	[0.996 0.816 0.594]	[663.996 23.150 31.898]
74	[0.476 0.013 0.353]	[673.476 22.585 31.349]
76	[0.975 0.902 0.596]	[661.976 24.899 36.1992]
231	[0.519 0.175 0.571]	[1078.519 39.474 53.764]
235	[0.168 0.973 0.767]	[1089.168 39.447 53.534]
257	[0.574 0.349 0.056]	[1115.574 43.477 62.122]
257	[0.824 0.633 0.669	[1148.824 39.897 60.948]
274	[0.452 0.375 0.975]	[1145.452 40.279 60.814]

Otro detalle interesante es que aunque se hable de convergencia de w, esto no es a una función concreta, si no a una familia de soluciones que cumple la propiedad de separar tales datos. Esto es notable en que ninguna de las w_f es igual, como ya adelantábamos en el apartado anterior.

Apartado 2.a.2 Ejecución para datos con ruido

Sabemos que ahora los datos no son separables, luego por más pasos que demos estos no convergerán.

Además como el ruido introducido es del 10% la precisión máxima a la que podemos aspirar es a 90%.

Como este algoritmo no es de regresión, no se está minimizando ningún valor, simplemente se iteran los datos y oscilamos entorno a la solución del apartado

anterior, pero sin que se llegue a detener, pues al no ser separables en ninguna épocas se dará que todos los datos estén bien clasificados.

Para observar esto mejor he planteado el siguiente experimento:

Manteniendo los vectores iniciales del apartado anterior variaremos el número de épocas y veremos el vector final y su precisión.

Observaremos que la precisión no guarda ninguna relación de proporcionalidad con el número de pasos, ya que por poner un caso, para el primer punto [0.574, 0.349, 0.057] empeora de 100 a 200 pasos pero vuelve a tener la misma precisión para 300 pasos.

Para max_iter = 100:

numero_pasos	w_0	w_f	Precisión (%)
100	[0.574, 0.349, 0.057]	[461.574, 29.602, 54.869]	86.0
100	[0.229, 0.664, 0.497]	[484.229, 28.285, 51.766]	86.0
100	[0.519, 0.175, 0.571]	[480.519, 28.202, 55.13]	86.0
100	[0.997, 0.817, 0.594]	[454.997, 29.599, 42.999]	82.0
100	[0.976, 0.902, 0.596]	[460.976, 22.188, 58.932]	83.0
100	[0.032, 0.094, 0.065]	[456.032, 4.774, 54.175]	76.0
100	[0.452, 0.375, 0.975]	[456.452, 10.36, 45.774]	79.0
100	[0.168, 0.973, 0.767]	[451.168, 29.135, 56.879]	85.0
100	[0.824, 0.633, 0.669]	[455.824, 25.473, 48.574]	86.0
100	[0.477, 0.013, 0.353]	[459.477, 0.021, 24.57]	77.0

Para max_iter = 200:

numero_pasos	w_0	w_f	Precisión (%)
200	[0.574, 0.349, 0.057]	[484.574, 21.757, 62.042]	82.0
200	[0.229, 0.664, 0.497]	[473.229, -1.77, 19.977]	75.0
200	[0.519, 0.175, 0.571]	[493.519, 23.984, 58.465]	83.0
200	[0.997, 0.817, 0.594]	[480.997, 26.531, 30.417]	86.0
200	[0.976, 0.902, 0.596]	[503.976, 4.218, 21.352]	81.0
200	[0.032, 0.094, 0.065]	[485.032, 1.791, 36.703]	74.0
200	[0.452, 0.375, 0.975]	[478.452, 23.616, 47.269]	86.0
200	[0.168, 0.973, 0.767]	[494.168, 33.638, 50.445]	82.0
200	[0.824, 0.633, 0.669]	[486.824, 24.497, 38.54]	86.0
200	[0.477, 0.013, 0.353]	[476.477, 23.059, 63.463]	83.0

Para max_iter = 300:

numero_pasos	w_0	w_f	Precisión (%)
300	[0.574, 0.349, 0.057]	[495.574, 27.744, 53.525]	86.0
300	[0.229, 0.664, 0.497]	[484.229, 20.829, 64.077]	81.0
300	[0.519, 0.175, 0.571]	[501.519, 28.838, 52.773]	86.0
300	[0.997, 0.817, 0.594]	[488.997, 6.238, 49.528]	76.0
300	[0.976, 0.902, 0.596]	[491.976, 25.317, 35.578]	88.0
300	[0.032, 0.094, 0.065]	[490.032, 22.965, 49.033]	86.0
300	[0.452, 0.375, 0.975]	[487.452, 30.27, 53.141]	86.0
300	[0.168, 0.973, 0.767]	[486.168, 28.835, 55.224]	86.0
300	[0.824, 0.633, 0.669]	[480.824, 15.16, 55.684]	80.0
300	[0.477, 0.013, 0.353]	[493.477, 35.702, 52.526]	82.0

Regresión logística

La regresión logística es un tipo de regresión lineal, puede entenderse como una variación del método de clasifición lineal donde la salida es una probabilidad (un valor entre cero y uno).

Por tratarse de un métdo de regresión lineal, disponenos de un error, el denominado error cruzado de entropía, el cual calculamos en la función `errorRegresionLofistica(x,y,x)` y que viene dado por

$$E_in(w) = \frac{1}{N}\sum_{n=1}^{N} \ln(1 + e^{-y_n w^T x_n})$$

Además su gradiente es

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

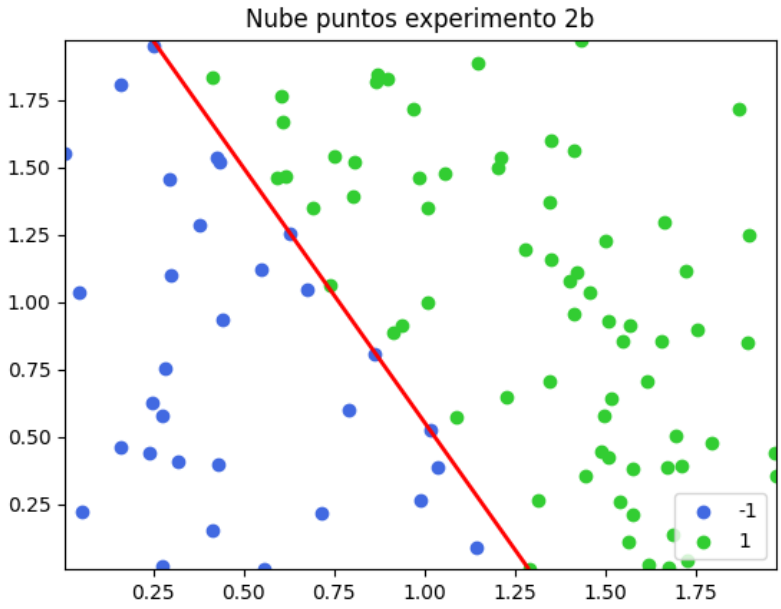
Para facilitar los cálculos, nosotros hemos utilizado $N = 1$ y este viene imprementado en la función `gradienteLogistica(x,y,w)`.

La implementación del algoritmo se encuentra en la función `regresionLogistica(x, y, tasa_aprendizaje, w_inicial, max_iteraciones, tolerancia)`

Experimento regresión logística

Los requisitos previos al experimento los puede encontrar bien explicados en la sección `##2b Experimento` del código (al rededor de la línea 574).

La nube de puntos, ya etiquetada y clasificada con la frontera determinada por la recta generada por dos puntos aleatorios es:



La función de clasificación construida tiene la forma (datos de la recta redondeados a dos decimales)

$f(x, y) = y + 1.891x - 2.444$

Vamos a ejecutar regresión logística para calcular la función solución g que aproxime a f, la tasa de aprencizaje es de $\eta = 0.01$ y el error permitido, la tolerancia del 0.01.

Un detalle es que como hemos construído nosotros el experimento, tenemos la certeza de que convergirá a la solución, luego la condición de parada de número máximo de iteraciones la he puesto *infinita* (como si no existitiera).

Tras 455 épocas el vector obtenido es $w = (-7.673, 6.516, 2.855)$, para el cual, si lo vemos en forma de recta (vuelvo a repetir razonamiento del apartado 2.a.1.a) se correspondería a $g(x, y) = y + 2.283x - 2.688$.

Así que en primera instancia, al ver que existe una diferencia considerable entre f y g ya sabremos que el ajuste no es perfecto.

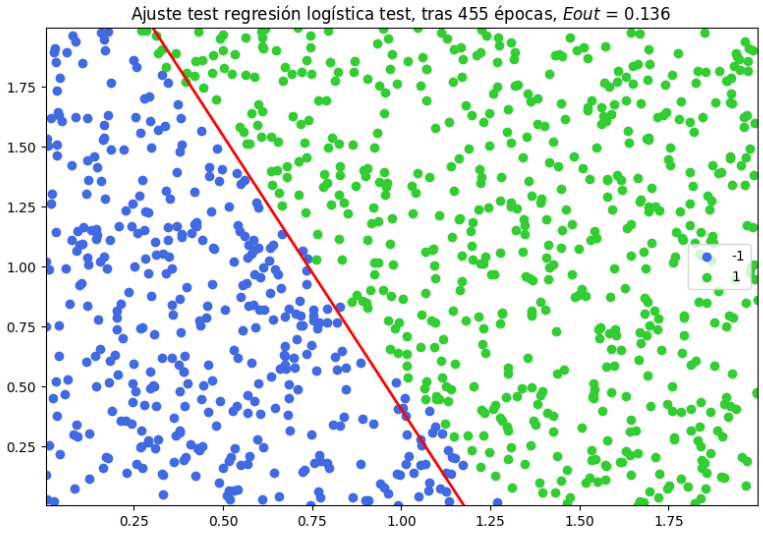
Confirmamos nuestra hipótisis gracias a los errores calculados:

$E_{in}(w) = 0.146$
$E_{out}(w) = 0.135$

Además para tener mayor intuición del significado del error de cruce de entropía hemos calculado la precición en el test:

```
Resultado clasificación para el test:
     Positivos fallados 7  de 605, lo que hace un porcentaje de 1.157
     Negativos fallados 17  de 395, lo que hace un porcentaje de 4.304
     Total fallados 24  de 1000, lo que hace un porcentaje de 2.4%
     La precisión es de 97.6 %
```

Veamos ahora un gráfico de los datos y la g:



Visualmente volvemos a corroborar que el ajuste no es perfecto.

Como conslusión obtenemos que aunque los datos sean separables y conozcamos la clase de funciones a la que pertenece la solución, nunca tendremos la certeza de haber calculado la función objetivo, solo tendremos una cota de error.

Esta conclusión proviene de la desigualdad de Hoeffding y de manera más genera de la cota de error de la dimensión de Vapnik-Chervonenkis:

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

Donde N es el tamaño de entrenamiento, d_{vc} la dimensión de Vapnik-Chervonenkis y δ la tolerancia.

Conocido esto podremos mejorar el margen de error: - Aumentando el tamaño de la muestra.
- Aumentando la tolerancia.

Sin embargo deberemos de tener presente que esta cota es teórica, es decir, no se está teniendo presente errores de redondeo introducidos por el ordenador. O incluso si nos ponemos estrictamente matemáticos si H toma valores reales, nunca jamás podremos representarlos (no existe un biyección entre los números enteros, los *representables con un ordernador* a los reales).

Finalmete corrobaremos que nuestro experimento inicial, del que partieron las deducciones; tiene valores significativos repitiendo el experimento 100 veces.

```
Resultados redondeados a tres decimales
El número medio de épocas es 458.06, con desviación típica 35.097
El E_out medio es 0.126, con desviación típica 0.011
La precisión media es 97.46, con desviación típica 1.342
```

Como vemos nuestro primer experimento presenta valores normales.

Bonus: Clasificación de dígitos

PLA-pocket

Como ya comenté en el algoritmo de perceptrón existen mejoras, como quedarse el mejor vector de pesos encontrado.

Como criterio de error usaré la precisión, a mayor precisión mejor será el w encontrado.

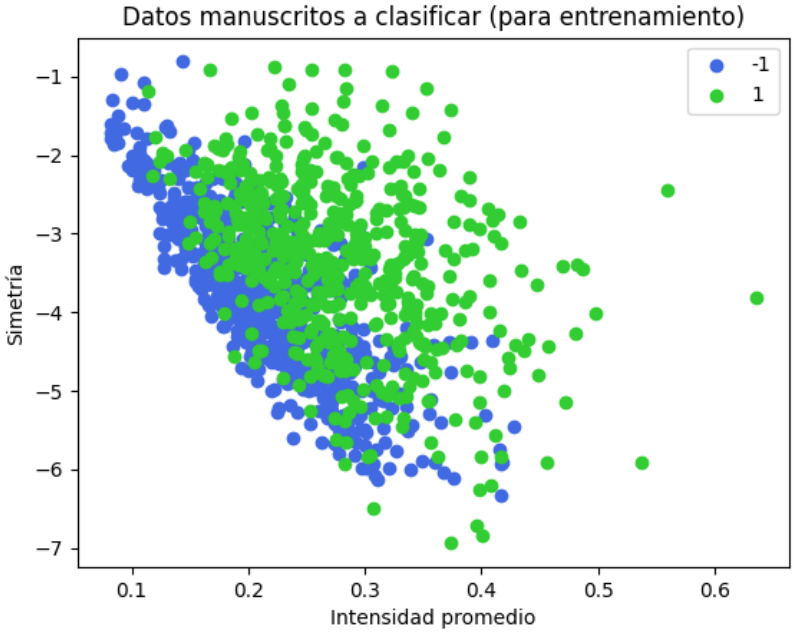
Problema a afrontar

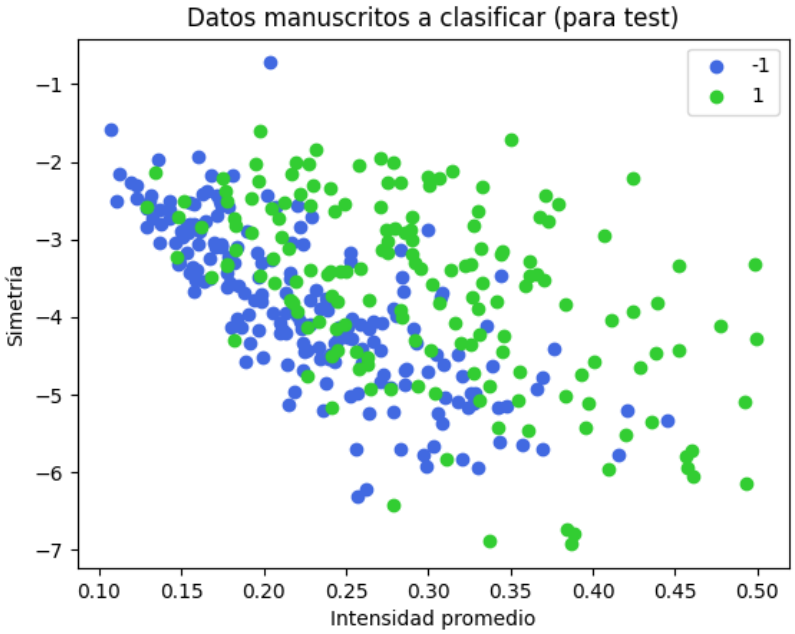
Se pretende clasificar los dígitos 4 y 8 a partir de las característica de intensidad promedio y simetría.

1. Planteamiento del problema de clasificación binaria.

Dado un conjunto de datos de entrenamiento, almacenando sus características de intensidad promedio y simetría en x_{test} y su etiqueta en y_{test} (dígito que corresponde).

(Las etiquetas negativas se corresponde al dígito cuatro y las positivas al 8).





Los modelos vistos en clase de clasificación son la clasificación lineal, la regresión lineal y la regresión logística.

Puesto que podemos suponer que los datos no son separables a priori optaría por un modelos de regresión lineal sin embargo ¿existiría alguna ventaja en usar el PLA o el PLA-Pocket? .

Voy a realizar un experimento previo analizando los distintos errores, usaré PLA y El error del SGD, aunque para PLA este error no está pensado, nos servirá para tener una referencia.

(Nota: Reutilizo código de la práctica primera, por eso no explico el algoritmo)

Experimento

Para $10, 20, 50, 100, 200, 500, 750, 1000$ iteraciones analizaré los resultados.

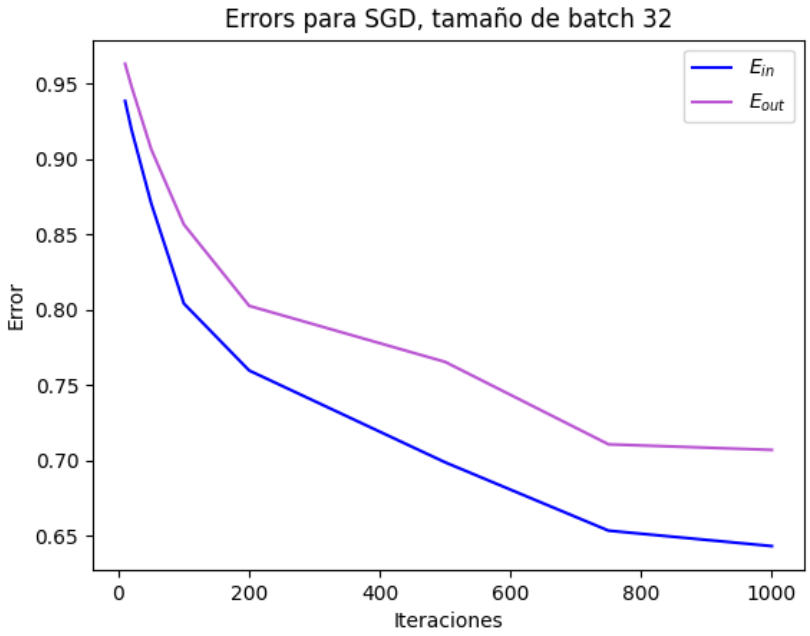
El algoritmo utilizado para regresión lineal es el SGD con tamaño de batch 32, porque en la práctica primera obuvimos resultados buenos con él.

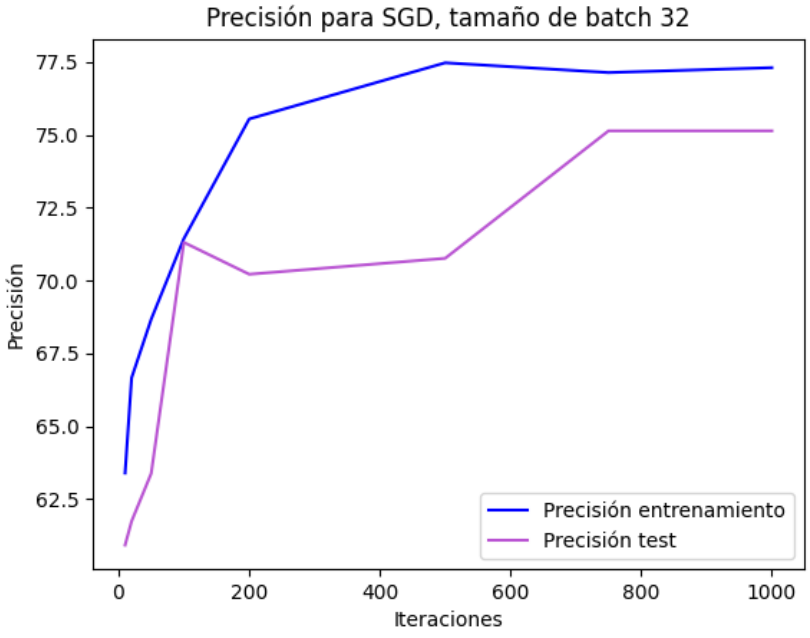
El resultado de las distintas ejecuciones es

SGD

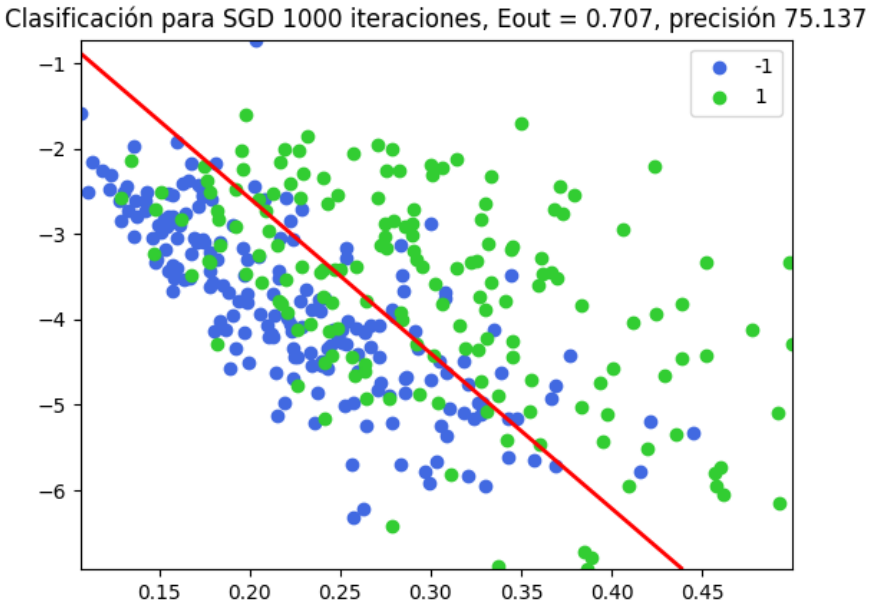
Iteraciones	Ein	Eout	Precision In	Precision out
10.0	0.939	0.963	63.4	60.929
20.0	0.919	0.948	66.667	61.749
50.0	0.871	0.907	68.677	63.388
100.0	0.804	0.857	71.441	71.311
200.0	0.76	0.803	75.544	70.219
500.0	0.699	0.765	77.471	70.765
750.0	0.653	0.711	77.136	75.137
1000.0	0.643	0.707	77.303	75.137

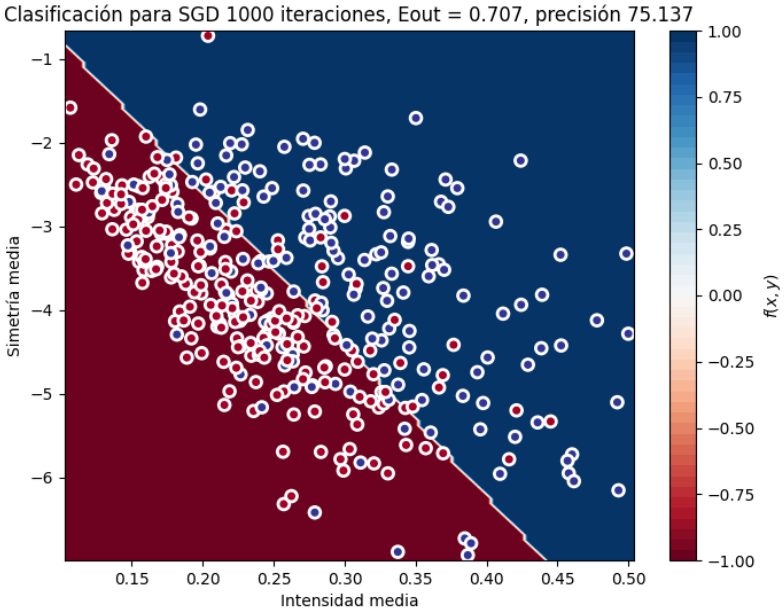
Los cuales se visualizan mejor





El ajuste quedaría de la forma:





NOTA ME HE DADP CUENTA DE UN ERROR DE ÚLTIMO MOMENTO PIDO POR FAVOR QUE LOS GRÁFICOS Y VALORES TOMADOS COMO

CIERTOS SEAN LOS DEL PROGRAMA Y LOS DE LA EJECUCIÓN (PORDON) LA

```
Clasificación para PLA

--- Pulsar tecla para continuar ---


--- Pulsar tecla para continuar ---

Iteraciones | Ein | Eout | Precision In | Precision out
---|---|---|---|---
10.0 | 234.301 | 233.089 | 61.223 | 60.656
20.0 | 129.905 | 125.85 | 72.278 | 73.497
50.0 | 185.352 | 179.928 | 74.121 | 75.137
100.0 | 162.984 | 161.412 | 76.801 | 75.137
200.0 | 342.869 | 332.132 | 69.095 | 69.945
500.0 | 342.587 | 331.843 | 69.095 | 69.945
750.0 | 342.328 | 331.587 | 69.095 | 69.945
1000.0 | 342.07 | 331.332 | 69.095 | 69.945

--- Pulsar tecla para continuar ---

Grafica para clasificación test PLA, con zona de clasificación
/home/blanca/repositorios/aprendizaje-automatico/practica2/ejercicio1.py:234: UserWarni
  ax.contour(XX,YY,fz(positions.T).reshape(X.shape[0],X.shape[0]),[0], colors='black')

--- Pulsar tecla para continuar ---


Grafica para clasificación test PLA, sin zona de clasificación

--- Pulsar tecla para continuar ---

Comparamos errores del test

--- Pulsar tecla para continuar ---

Clasificación para PLA-pocket

--- Pulsar tecla para continuar ---


--- Pulsar tecla para continuar ---

Iteraciones | Ein | Eout | Precision In | Precision out
```

```
---|---|---|---|---
10.0 | 6.963 | 7.408 | 76.884 | 74.863
20.0 | 6.963 | 7.408 | 76.884 | 74.863
50.0 | 102.367 | 102.372 | 77.136 | 75.41
100.0 | 102.367 | 102.372 | 77.136 | 75.41
200.0 | 102.367 | 102.372 | 77.136 | 75.41
500.0 | 102.367 | 102.372 | 77.136 | 75.41
750.0 | 102.367 | 102.372 | 77.136 | 75.41
1000.0 | 102.367 | 102.372 | 77.136 | 75.41

--- Pulsar tecla para continuar ---

Grafica para clasificación test PLA_POCKET, con zona de clasificación
/home/blanca/repositorios/aprendizaje-automatico/practica2/ejercicio1.py:234: UserWarning: No cor
  ax.contour(XX,YY,fz(positions.T).reshape(X.shape[0],X.shape[0]),[0], colors='black')

Gráfica para clasificación test PLA_POCKET, sin zona de clasificación

--- Pulsar tecla para continuar ---

Fin experimento previo
Ajusto primero usando SGD
eta = 0.01,  error 0.01 y máximo iteraciones 50
w obtenido = [[0.41013082]
 [1.3563353 ]
 [0.20352679]]

--- Pulsar tecla para continuar ---

__ Análisis de los resultados__
Ein_SGD = 0.8709668929832045
Eout_SGD = 0.9066741855472633
accuracy_in_SGD = 68.7604690117253
accuracy_out_SGD = 63.66120218579234

--- Pulsar tecla para continuar ---

Procedemos a concatenar al w conseguida con SGD con el Pla-pocket
con un máximo de 50  iteraciones
w obtenido = [-7.58986918 98.34965813  4.89202679] tras 50 epocas

--- Pulsar tecla para continuar ---

__ Análisis de los resultados__
Ein_PLA_POCKET = 41.757121355516595
Etest_PLA_POCKET = 44.10136255980877
```

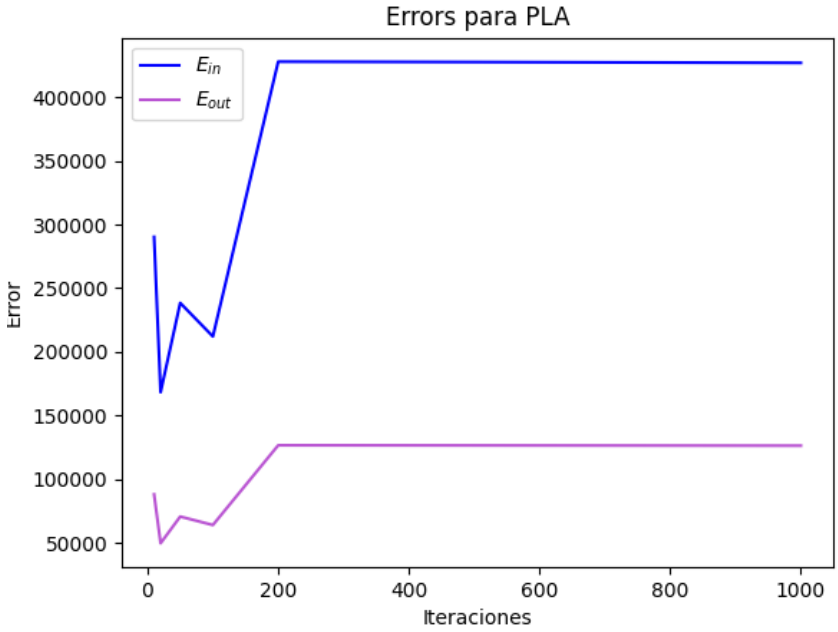
```
accuracy_in_PLA_POCKET = 77.21943048576215
accuracy_test_PLA_POCKET = 73.77049180327869

--- Pulsar tecla para continuar ---

Análisis de la cota
N = 366, H = 55340232221128654848, dvc = 3, delta = 0.05, E_test = 44.10136255980877
Usando desigualdad de Hoeffding E_out <= 44.36048284355012
Usando la generalización VC E_out <= 44.82819968092755
```

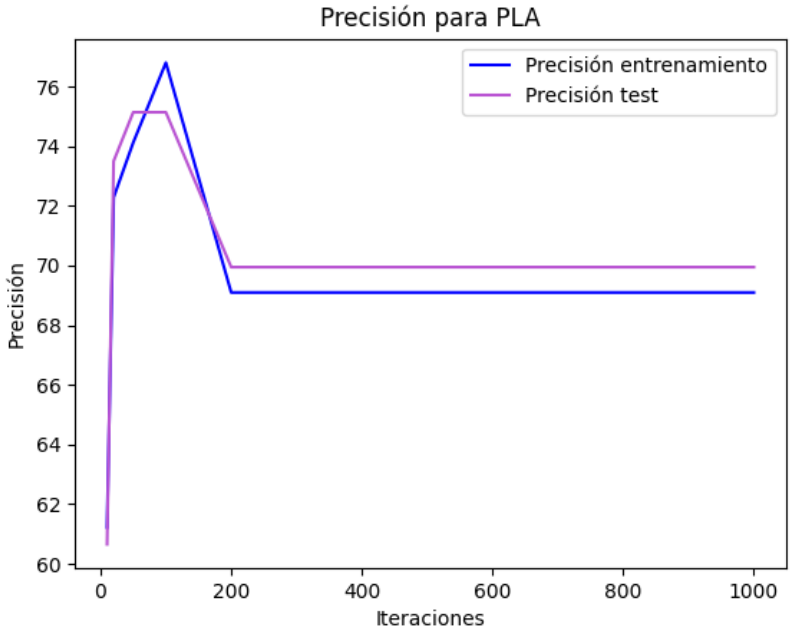
PLA

Clasificación para PLA

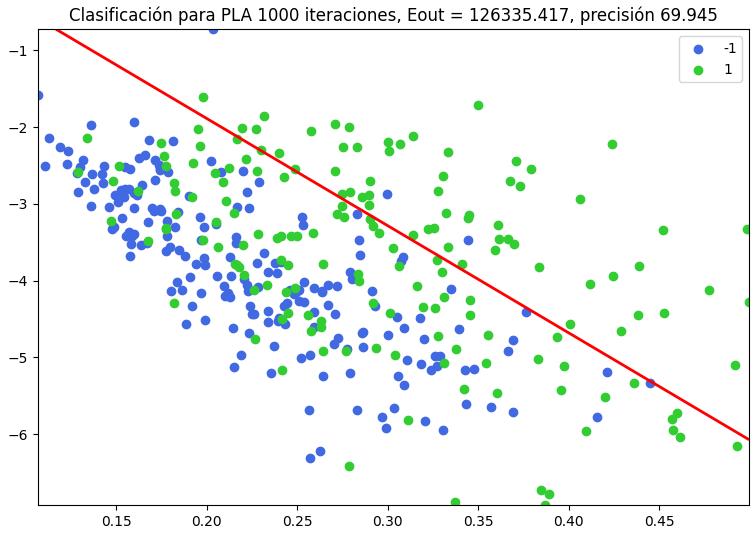
Iteraciones	Ein	Eout	Precision In	Precision out
10.0	290299.348	88097.573	61.223	60.656
20.0	168336.859	49696.562	72.278	73.497
50.0	238436.31	70641.13	74.121	75.137
100.0	212128.357	64003.368	76.801	75.137
200.0	427982.958	126626.179	69.095	69.945
500.0	427658.949	126524.163	69.095	69.945
750.0	427346.177	126429.767	69.095	69.945
1000.0	427033.546	126335.417	69.095	69.945

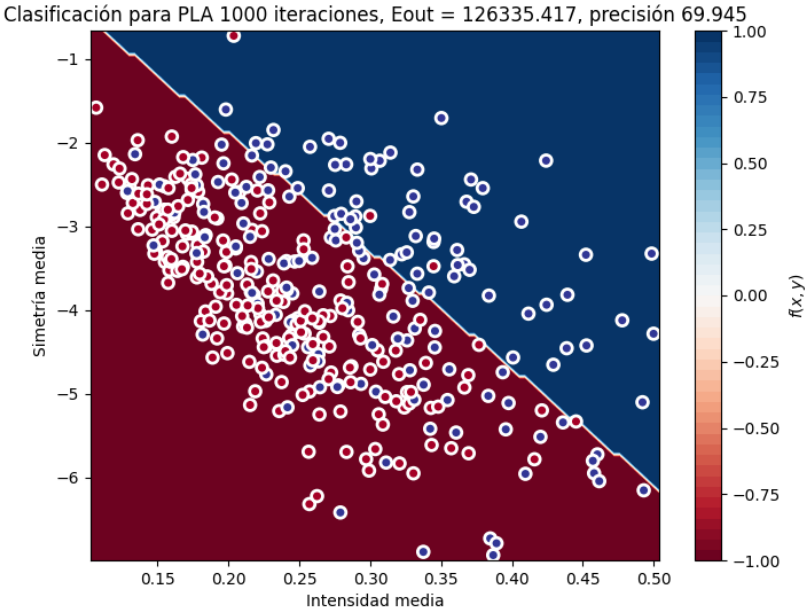
Los cuales se visualizan mejor





El ajuste quedaría de la forma:





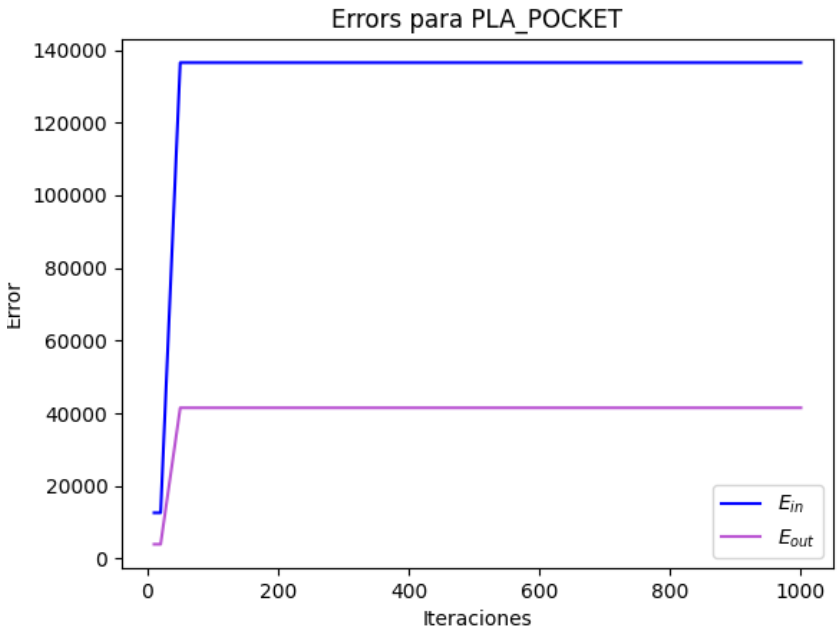
Como víamos en los ejercicios anteriores las soluciones oscilan independiente de su error y precisión.

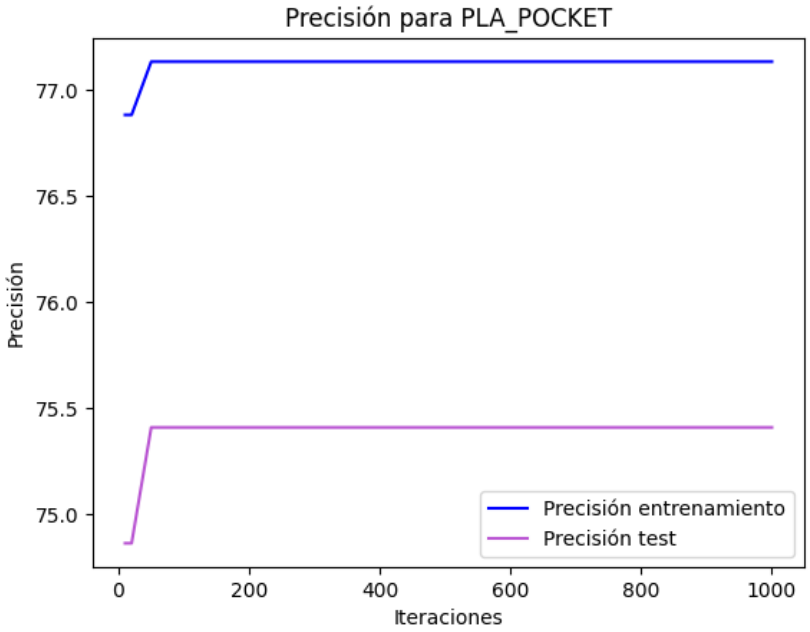
PLA_POCKET

Clasificación para PLA-pocket

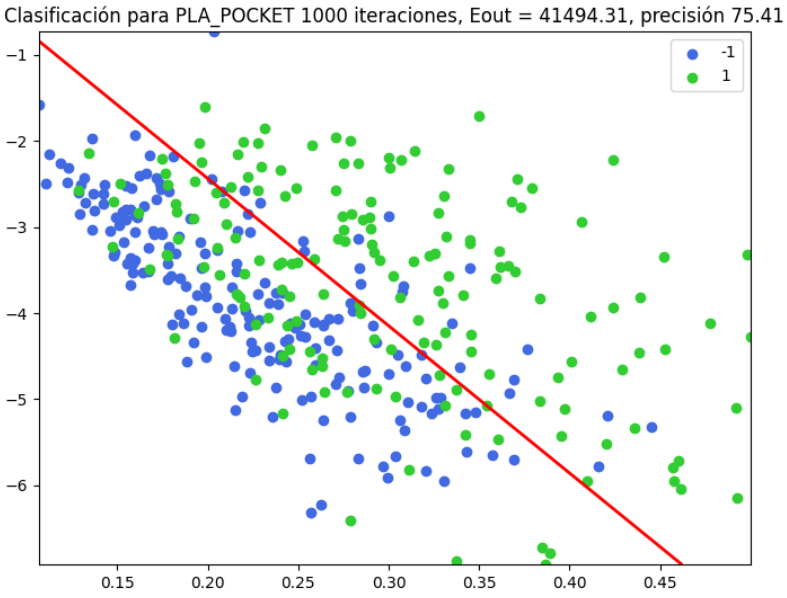
Iteraciones	Ein	Eout	Precision In	Precision out
10.0	12590.265	3934.544	76.884	74.863
20.0	12590.265	3934.544	76.884	74.863
50.0	136518.098	41494.31	77.136	75.41
100.0	136518.098	41494.31	77.136	75.41
200.0	136518.098	41494.31	77.136	75.41
500.0	136518.098	41494.31	77.136	75.41
750.0	136518.098	41494.31	77.136	75.41
1000.0	136518.098	41494.31	77.136	75.41

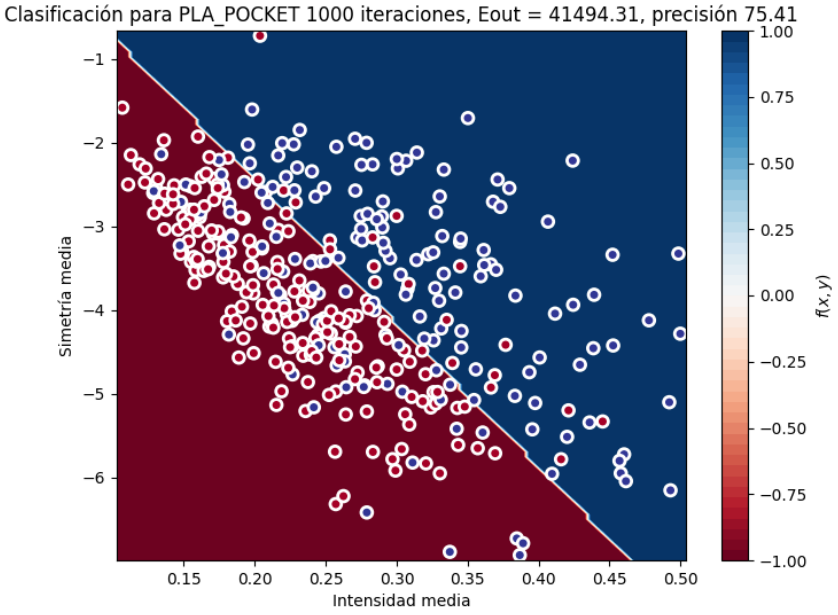
Los cuales se visualizan mejor





El ajuste quedaría de la forma:





En este caso la precisión se mantiene siendo incluso superior a la del SGD.

Puesto que PLA-Pocket tiene mejores resultados pero su coste computacional es mayor y hemos visto en los algoritmos basados en perceptrón el vector inicial es crucial, este experimento preliminar nos conduce a pensar que un buen modelo de búsqueda es acercarnos a una solución con SGD y mejorarla con pocket PLA.

Experimento

Utilizaré 50 iteraciones para el SGD y tras esto otras 50 para el PLA - Pocket y veremos si esta técnica supera a las 100 iteraciones de cualquiera de los métodos de forma individual.

```
__ Análisis de los resultados__
Ein_SGD = 0.8709668929832045
Eout_SGD = 0.9066741855472633
accuracy_in_SGD = 68.7604690117253
accuracy_out_SGD = 63.66120218579234
```

```
Procedemos a concatenar al w conseguida con SGD con el Pla-pocket
con un máximo de 50  iteraciones
w obtenido = [-7.58986918 98.34965813  4.89202679] tras 50 epocas
```

```
__ Análisis de los resultados__
Ein_PLA_POCKET = 59581.08303825323
Etest_PLA_POCKET = 18945.58756763352
accuracy_in_PLA_POCKET = 77.21943048576215
accuracy_test_PLA_POCKET = 73.77049180327869
```

(Nota: Aunque con el pocket no se minimiza el Ein, me extraña que el error Ein Etest sea tan grande, ya que guarda cierta correlación con la precisión. Me da que pensar que se me está escapando algún detalle en los cálculos)

El pla pocket individual ha obtenido mejores resultados, luego como conclusión no utilizaría nunca el SGD solo ni tampoco el PlA convencional, pero sin embargo el PLA-Pocket tiene un coste computacional mayor.

Así que probablemente el modelo más adecuado sea el combiando SGD y PLA convencional, pero todo dependerá del problema.

Obtención de cotas

Para acotar el error fuer de la muestra E_{out} usaré la ya mencionada desiguardad de Hoeffing, con los errores E_{test} y E_{out}. La tolerancia o nivel de confiaza es de $\delta = 0.05$ (es decir ciertas con una probabilidad de al menos 0.95).

Además el tamaño $|H|$ proviene de discretizar el problema a float de 64 bits $|\mathcal{H}| = 32^{64}$

$$E_out(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N} \ln \frac{2|H|}{\delta}}$$

(Sale muy grande debido a E_{test})

Alternativamente podríamos usar la cota de VC

Para el caso perceptrón $d_{vc} = 3$

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

```
Análisis de la cota
N = 366, H = 55340232221128654848, dvc = 3, delta = 0.05, E_test = 18945.58756763352
Usando desigualdad de Hoeffding E_out <= 18945.846687917263
Usando la generalización VC E_out <= 18946.314404754638
```

La cota más informativa es la del test (más que si hubieéramos hecho la de E_in, ya que es independiente de los datos utilizados durante el aprendizaje.

Respecto si usar VC o Hoeffding, la cota menor es la más precisa, ya que ambas están avaladas por la teoría.