Práctica 2

Aprendizaje Automático

Blanca Cano Camarero

May 1, 2021

Índice

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.

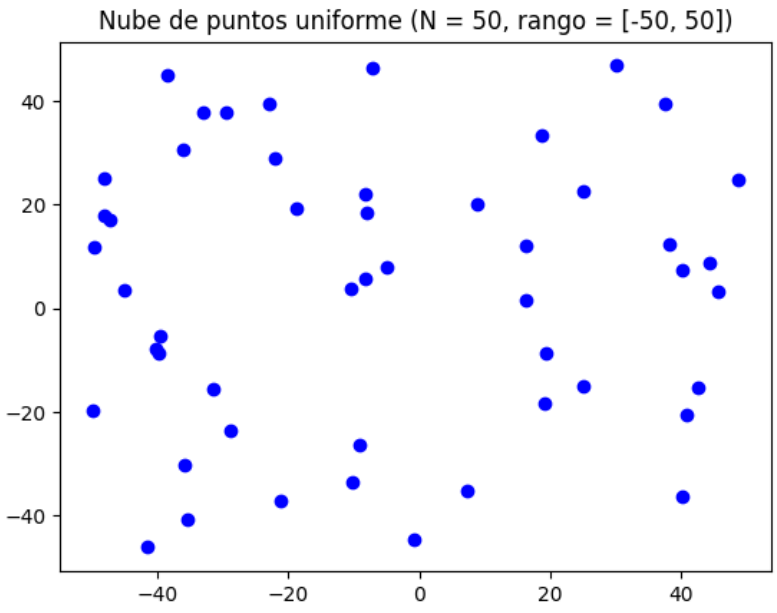
- `simula_recta(intervalo)` , que simula de forma aleatoria los parámetros, v = (a, b) de una recta, $y = ax + b$, que corta al cuadrado $[-50, 50] \times [-50, 50]$.

1 Dibujo de las gráficas

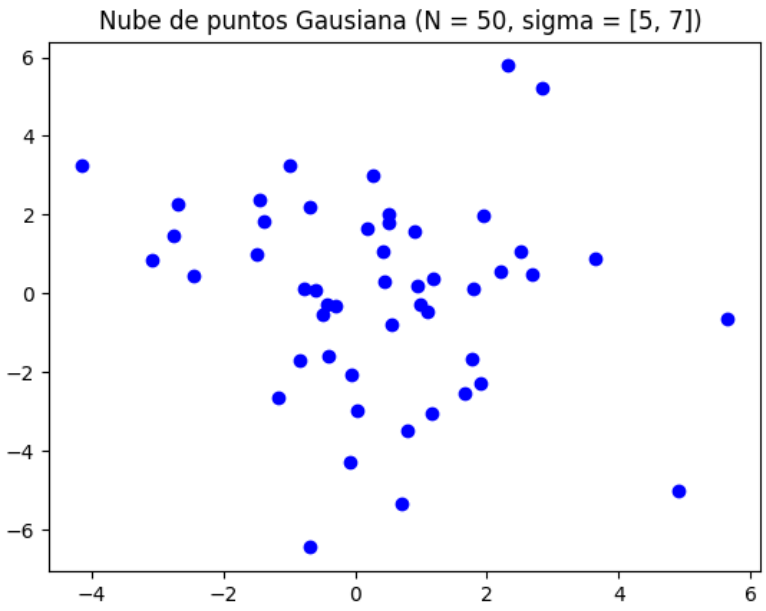
Dibujar gráficas con las nubes de puntos simuladas con las siguientes condiciones:

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
    '''
```

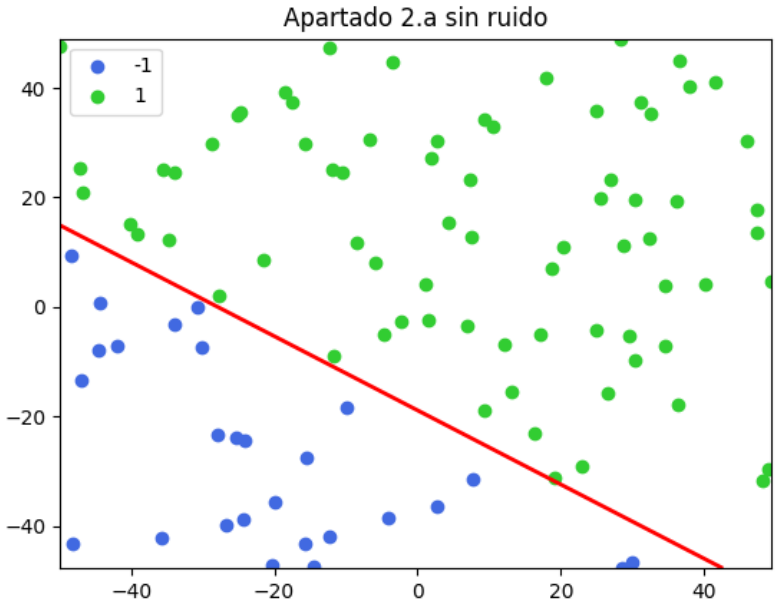
Está implementadas con las funciones `scattered` y `contour` de la librería `matplotlib.pyplot`.

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).

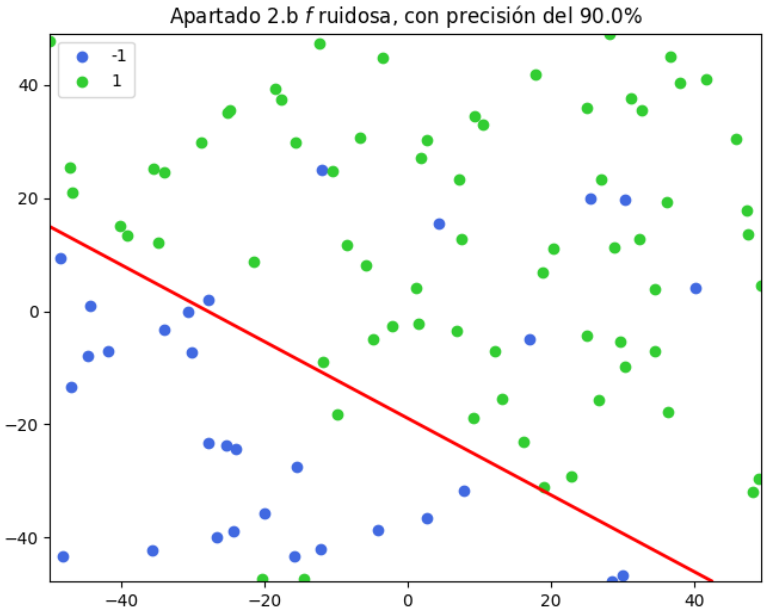
Sin embargo se ha optado por hacer uso de la función `contour` para tener mayor generalidad, ya que esta es capaz de pintar los puntos de cualquier ecuación $g(x, y) = 0$ con $g : \mathbb{R}^2 \longrightarrow \mathbb{R}$ cualquiera.

Etiquetado sin ruido.

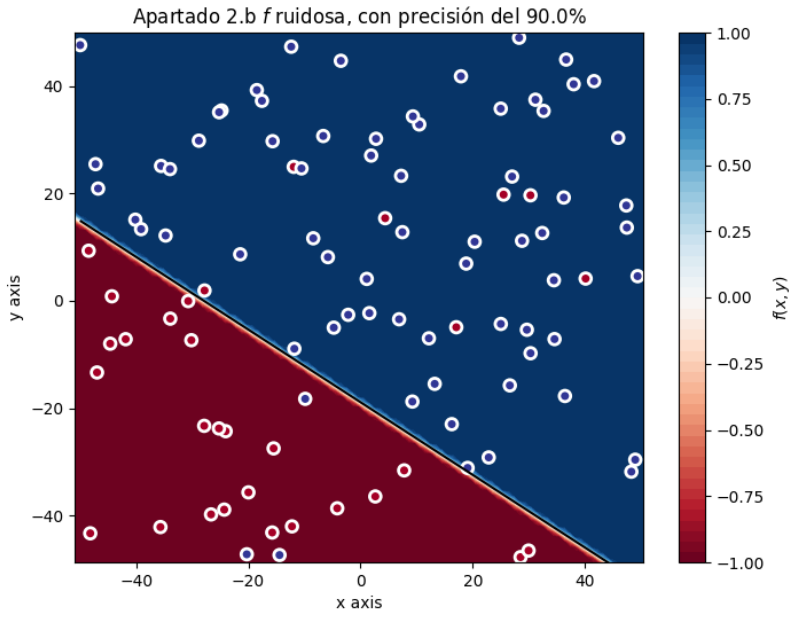


Como podemos ver todo está bien clasificado.

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:

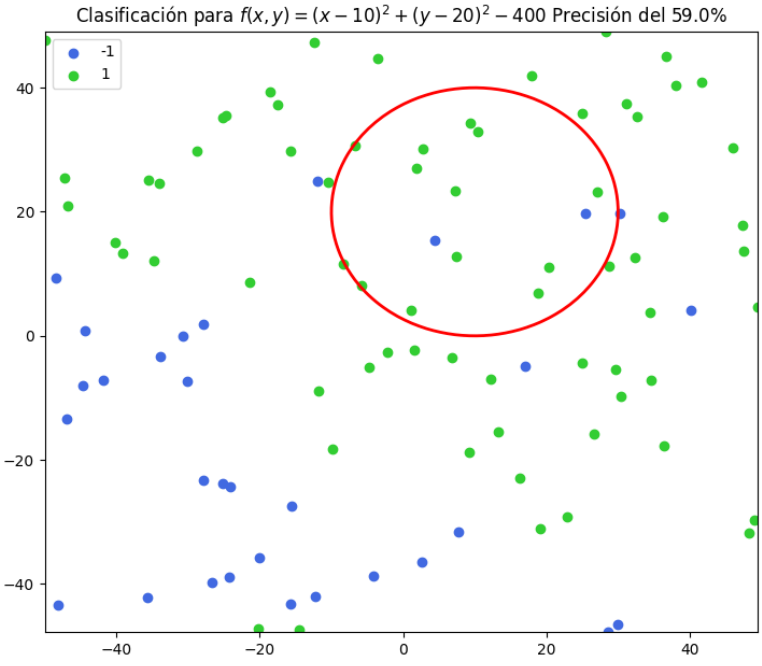
Para analizar la bondad del ajuste vamos a tener en cuenta la precisión, además para comprobar si beneficia más a un tipo u a otro analizando los positivos y negativos fallados.

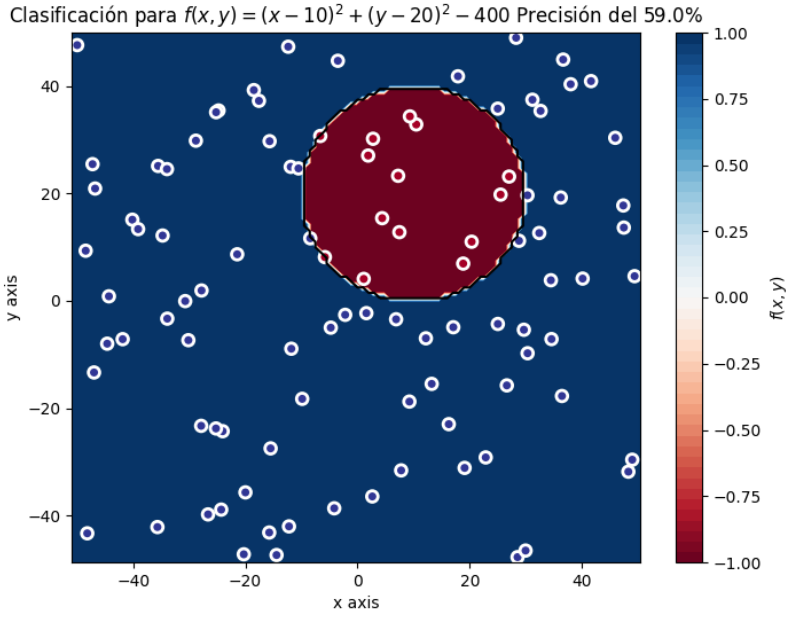
Recordamos que la precisión se define como $\frac{\text{datos bien clasificados}}{\text{datos bien clasificados}}$, nosotros además la indicamos como porcentaje, multiplicada por 100.

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%

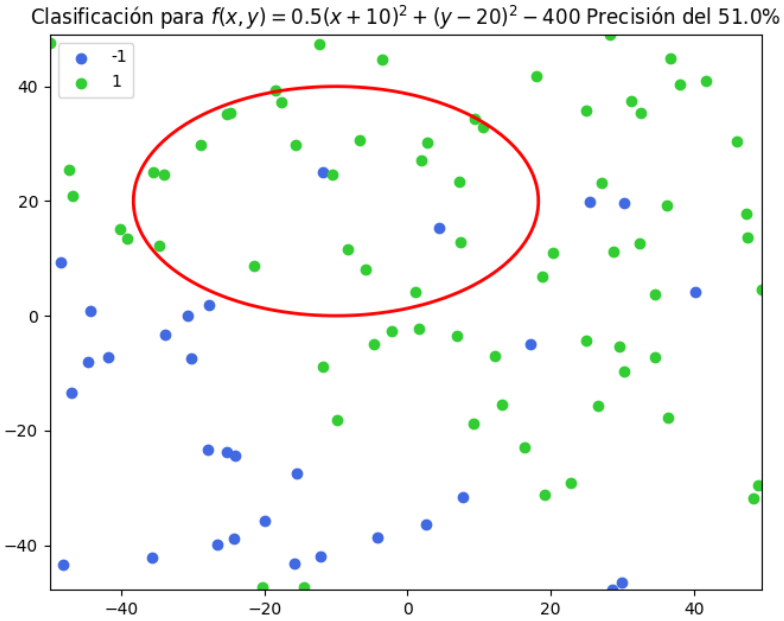


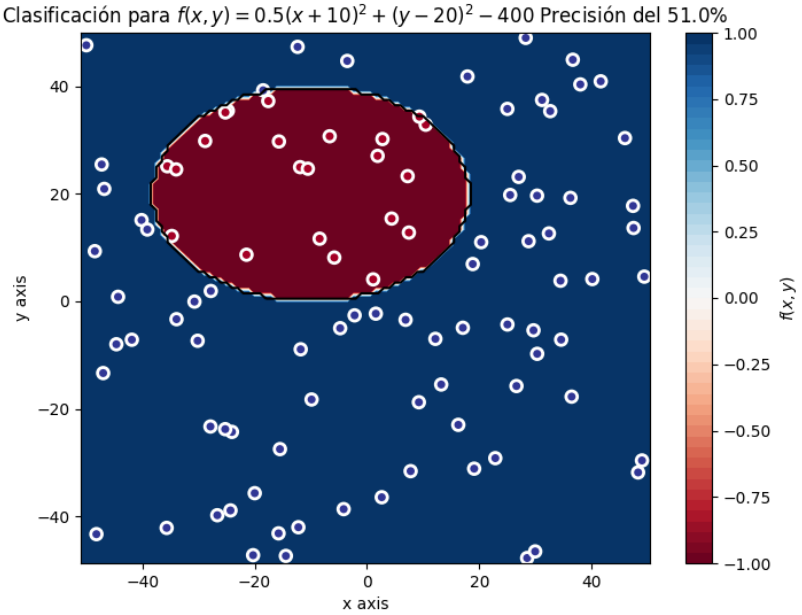


Este caso empeora la precisión y vemos que falla la mayoría de los negativos.

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

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

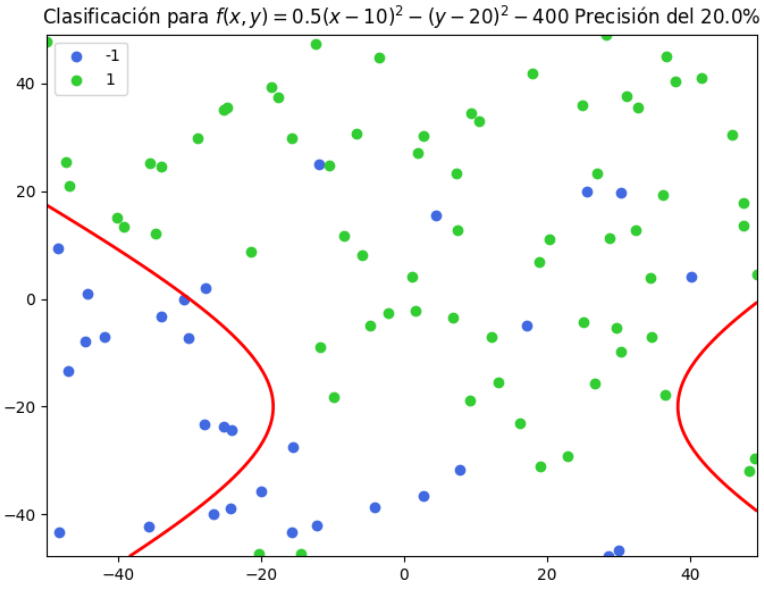


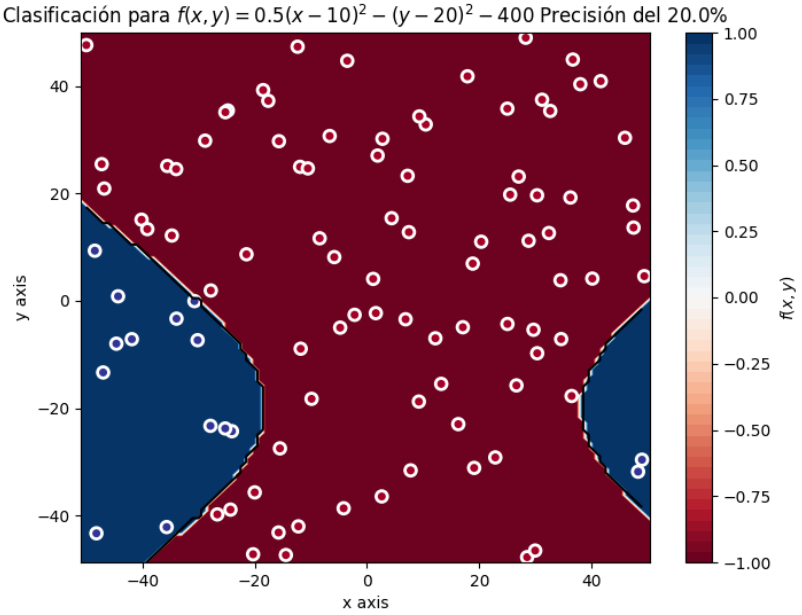


Parecido al ajuste anterior, empeorando la clasificación.

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

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

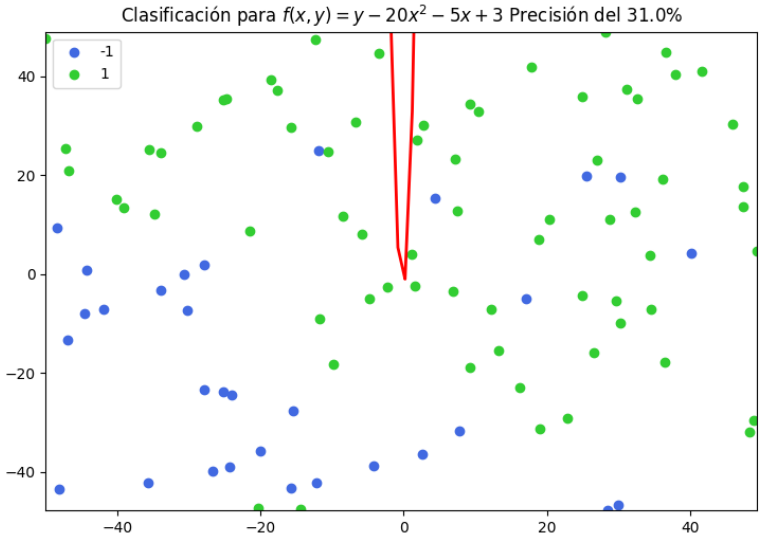


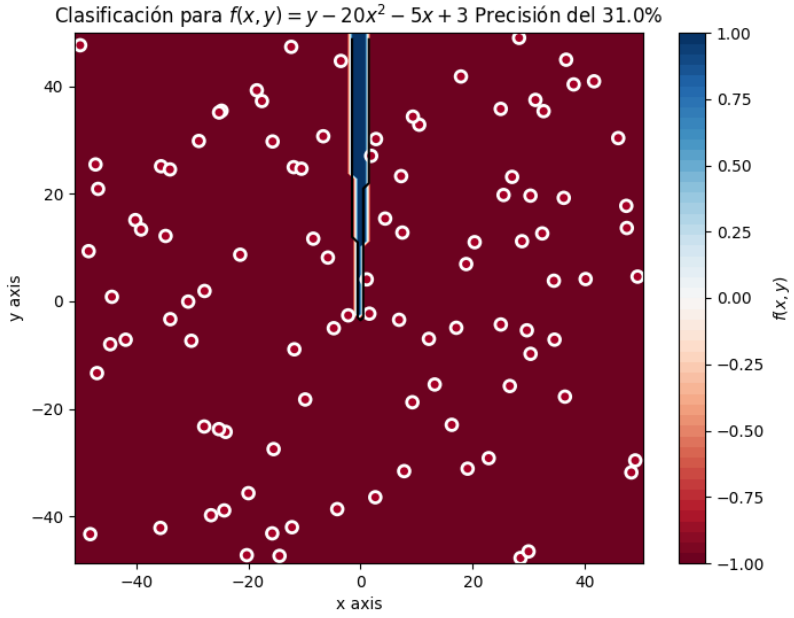


En este caso falla la mayoría de los negativos y es la peor de los ajustes, curiosamente, por la forma que tiene si hubiéramos clasificado con la opuesta $-f(x,y)$, los resultados hubieran sido mucho mejor, un 80% de precisión, lejos aún del 90% inicial.

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

- 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

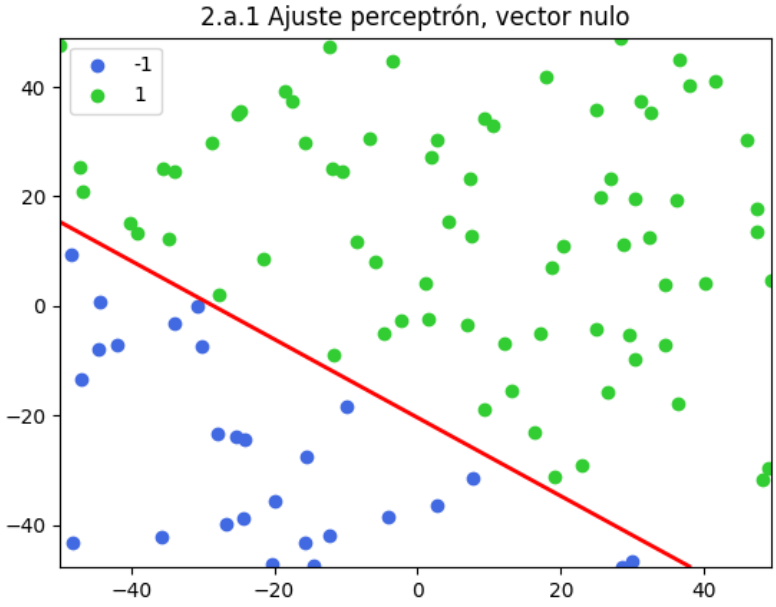
Los coeficientes del vector final son $[661, 23.202, 32.391]$, se ha encontrado tras 75 épocas, antes del máximo de pasos establecido. Esto es gracias a que los datos eran separables y se ha encontrado una solución.

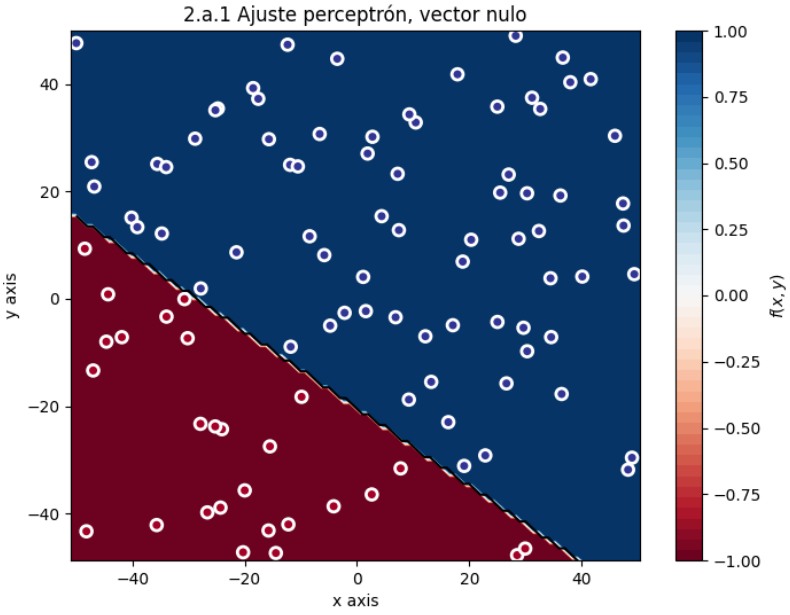
La solución no tiene porqué ser única, de hecho los coeficientes de la recta objetivo original eran $y = ax + b$ con $ a =-0.677, b = -18.89$

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 $w_1 + xw_2 + yw_3 = 0$ $y = \frac{w_2}{w_3}x + \frac{w_1}{w_3}$

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

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.

Apartado 2.a.2

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. 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 y no guarda ninguna relación de proporcionalidad con el número de pasos, ya que 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

numero_pasos	w_0	w_f	Precisión (%)
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

Bonus: Clasificación de dígitos

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).

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 voy a optar por un modelos de regresión lineal.

Reutilizo código de la práctica primera.

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.